Boa ideia! Utilizando recursos de fala: Faça seu sistema obedecer sua voz

Clube Delphi

Ano 13 :: Edição 153 :: R$ 14,90

DevMedia

Componentes para FireMonkey
Conheça o que a TMS pode oferecer

Desvendando métodos anônimos
Explore esse recurso pouco conhecido

Tudo sobre padrões estruturais
Melhore a arquitetura de seu sistema

dbExpress para FireDac

Migração passo a passo

ISSN 1517990-7

VOCÊ É ASSINANTE MVP?

ENTÃO PODE TER CERTEZA: O MERCADO PREFERE VOCÊ!

DESENVOLVEDOR ATUALIZADO É O MAIS VALIOSO DO MERCADO!

Muitos profissionais estão acomodados, não se atualizam, não leem livros e não vão a eventos. Acham que é a empresa que deve mantê-los atualizados. Gastam dinheiro com tudo, mas acham caro investir menos de R$60,00 por mês em suas carreiras.

Mas você não! Você é MVP! Parabéns!

TENHA ACESSO A:

- +DE 260 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 7.850 VÍDEO-AULAS

POR APENAS 59,90 MENSAIS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

DevMedia

Acesse: www.devmedia.com.br/mvp

Sumário

Artigo no estilo Solução Completa

06 – Migrando de dbExpress para FireDac
[*Luciano Pimenta*]

Conteúdo sobre Novidades

13 – TMS Pack for FireMonkey
[*Fabrício Hissao Kawata*]

Conteúdo sobre Boas Práticas

22 – Desvendando métodos anônimos
[*Filipe Dalepiane*]

Artigo do tipo Mentoring

30 – Padrões estruturais em Delphi
[*Guinther Pauli*]

44 – Uso de voz em Delphi
[*Vanderson Cavalcante Freitas*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 13 - 153ª Edição 2013 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Wesley Yamazack (yamazack@gmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

Em um mundo onde cada vez mais o BYOD (Bring Your Own Device) cresce, onde funcionários são estimulados a utilizar seus próprios dispositivos, é cada vez mais imperativo construir sistemas que sejam compatíveis com as mais diversas plataformas. Mas, o que fazer com o grande volume de aplicativos existentes que são presos a uma única plataforma/banco de dados? Pensando justamente nessa situação, a Embarcadero trouxe para os desenvolvedores Delphi uma nova plataforma de acesso a dados, o FireDac. Esta é multi banco e sim, multi plataforma. Atualmente sendo compatível com iOS. Mas para utilizar essa nova ferramenta existe uma certa curva de aprendizado, curva essa que será amenizada pelo artigo do Luciano Pimenta, que mostra como começar uma migração a partir de um sistema simples mas real.

Ainda sobre multi plataforma, também existe a dependência visual da VCL para ambientes Windows. Isso já foi solucionado com a criação do FireMonkey. Contudo, o que tem prejudicado sua adoção é falta de componentes prontos. O Fabrício traz uma novidade para aqueles que tem tentado adotar o FireMonkey, um artigo sobre o pacote TMS para FireMonkey. Vários controles personalizados e explicados, não perca.

Com todas essas novidades ainda temos recursos inexplorados ou, que não chamam a atenção da comunidade. Porém sua utilização traz flexibilidade e qualidade para o código. O Filipe apresenta nesta edição um artigo que explora, conceitua e aplica o uso de métodos anônimos em Delphi. Se você ainda não sabe o que é isso, esse é o seu momento de subir seu nível de desenvolvimento. Seu uso pode ser alinhado com outras boas práticas que sempre estão por aqui em nossa revista.

Padrões de Projeto. Seu entendimento, conceituação e aplicação são essenciais para quem busca um software que responda rápido às mudanças impostas, sejam por alteração de requisitos ou atualização de componentes de tecnologia (frameworks, SOs). O Guinther traz em seu artigo uma verdadeira referência para os padrões Estruturais.

Com um código bem estruturado, flexível, fica fácil adicionar inovação. O Vanderson apresenta algo inusitado para sistemas comerciais: a utilização por voz. Nesse artigo é mostrado como implementar essa funcionalidade em seus projetos, permitindo o uso dos mesmos através da fala.

E por fim, a Leandra mostra em uma solução completa como isolar um problema comum e tirar proveito de sua solução. Ela demonstra como centralizar a geração de relatórios para posteriormente publicá-la via web service a qualquer aplicativo cliente, independentemente da plataforma.

Temos vários assuntos nesta edição, vários deles curiosos e importantes. Dessa forma, boa leitura!

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

DevMedia

Seu **conhecimento** tem **visibilidade** na DevMedia

Publique artigos e vídeos no site e ganhe mais visibilidade em seu trabalho. Somos o maior site do Brasil para desenvolvedores.

Para publicar é bem simples: mande um email para nosso editor **Wesley Yamazack (yamazack@gmail.com)**.

Para mais informações acesse:
http://www.devmedia.com.br/seja-um-autor-devmedia/28103

DevMedia

Migrando de dbExpress para FireDAC

Dicas para migração

Resumo DevMan

Migrando de dbExpress para FireDAC:

Vamos abordar neste artigo como migrar para a mais nova suíte de acesso a dados da Embarcadero, o FireDAC. Utilizaremos para isso uma solução baseada em dbExpress, na qual permitirá uma visualização mais próxima da realidade. Conheceremos no decorrer do processo algumas facilidades, dificuldades, dicas e tudo para que você possa migrar sua solução sem grandes problemas.

Em que situação o tema é útil:

O tema é útil em casos de aplicações desenvolvidas em cima do framework dbExpress na qual o foco é um processo de migração de tecnologia. Outra hipótese é para quem está iniciando aplicações que fazem uso de banco de dados e querem suportar múltiplos bancos de dados em diferentes versões através de um único framework. Explorar os recursos do FireDAC e do impacto das mudanças como um aspecto positivo.

A partir do Delphi XE3 temos um novo framework de acesso a dados, o FireDAC. O FireDAC é o antigo AnyDAC da DASoft Technologies que foi adquirido pela Embarcadero e está presente para usuários registrados das versões Enterprise, Ultimate ou Architect.

Para quem usa a versão Professional, pode adquirir um pacote add-on client/server. O FireDAC possui uma engine otimizada para acesso a dados, não possui driver para ser distribuído (arquivo DLL, por exemplo).

Existem várias funções para abstrair o servidor de banco de dados usado, assim, sua aplicação pode facilmente ser multibanco. O servidor de banco de dados fica a cargo de sua escolha ou do seu cliente.

O dbExpress tinha também esse objetivo, suportar aplicações multibancos, mas exemplos indicam que projetos que utilizam mais de um servidor de banco, usando dbExpress, não tem toda essa facilidade. FireDAC por sua vez, veremos que facilita esse trabalho e ainda tem a vantagem de acesso a banco de dados no iOS. Os principais bancos suportados são:

- MySQL;
- SQL Server;
- Oracle;
- InterBase;
- PostgreSQL;
- SQLite;
- Sybase SQL Anywhere;
- Access;
- DB2;
- Firebird.

No XE4, existe o suporte no iOS para os seguintes bancos de dados:

- InterBase – InterBase ToGo and IBLite;
- SQLite.

O projeto a ser migrado

Vamos conhecer o projeto que utiliza o dbExpress e será migrado para FireDAC. Esse projeto é um pequeno sistema client/server para uma oficina mecânica, onde temos cadastros de clientes, veículos, orçamentos, ordens de serviço e peças. A aplicação possui consultas auxiliares e utiliza o dbExpress como framework de acesso a dados. Na **Figura 1** temos o DataModule.

No lado esquerdo estão os componentes para cadastros. No exemplo temos o trio de componentes (SQLDataSet + DataSetProvider + ClientDataSet) padrão para a estrutura do dbExpress.

Do lado direito temos as consultas auxiliares. Como estamos trabalhando apenas com consultas, não precisamos usar um trio para cada consulta. A técnica consiste em usar um SQLDataSet e um DataSetProvider e um ClientDataSet para cada consulta.

Precisamos configurar o DataSetProvider para indicar que a consulta (código SQL) fica no ClientDataSet. A propriedade Option>poAllowCommandText deve ser configurada para true. O código SQL fica no ClientDataSet na propriedade CommandText.

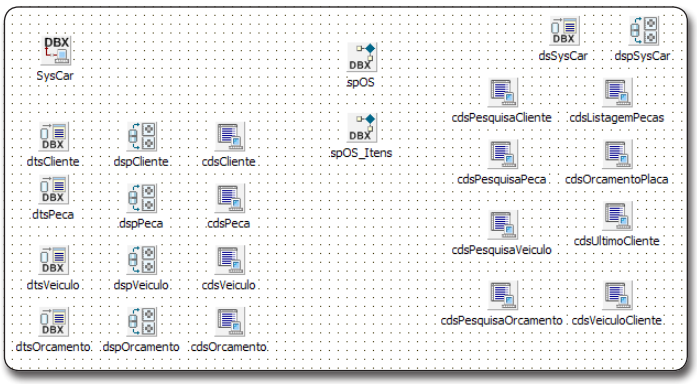

Figura 1. DataModule da aplicação

Isso é uma facilidade para usar o ClientDataSet (dados em memória). Note que temos dois controles centralizados que são ligados em Stored Procedures (SQLStoredProc).

Padronização

Para as telas usamos a técnica de herança visual de formulários. Veja na **Figura 2** o exemplo da tela base de cadastro.

Temos uma aba para os campos do cadastro e outra para pesquisar os registros do referido cadastro. A consulta dessa tela e consultas auxiliares são preenchidas com os controles (ClientDataSet) comentados anteriormente. Veja na **Figura 3** o cadastro de cliente em execução.

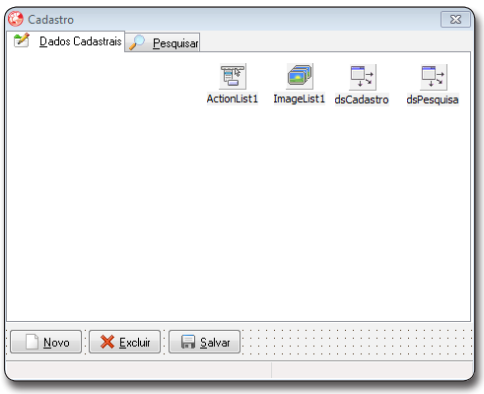

Figura 2. Tela de cadastro base

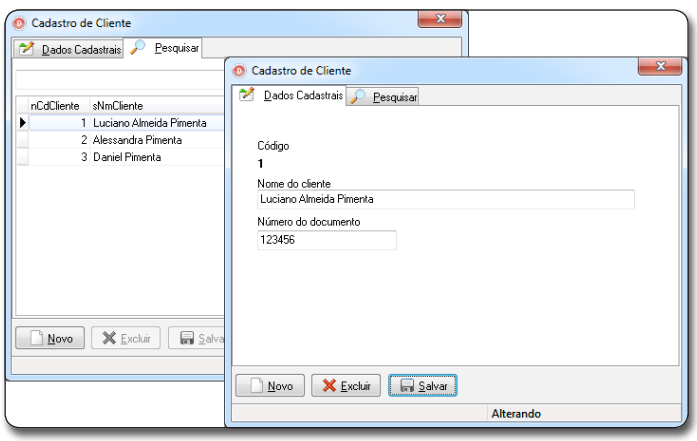

Figura 3. Cadastro de clientes

Outra funcionalidade importante que temos é o cadastro de Ordem de serviço. No cadastro temos informações master/detail, pois, temos os dados da OS e as peças usadas na mesma. Veja na **Figura 4** o cadastro em ação.

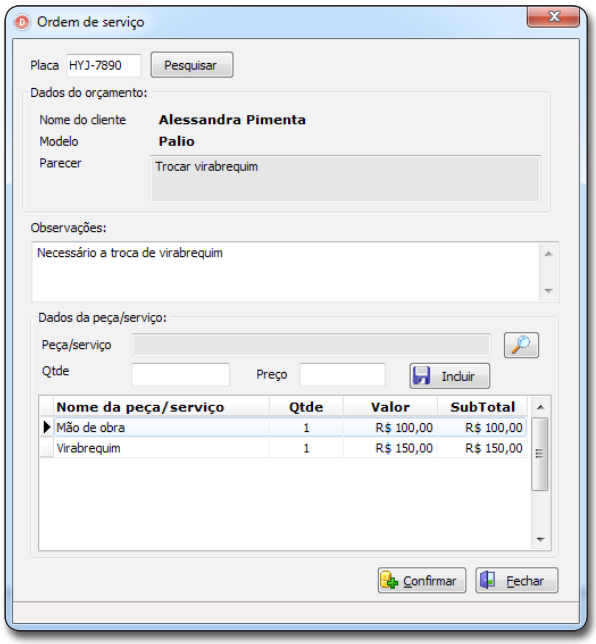

Figura 4. Cadastro da Ordem de serviço

Esse cadastro é interessante, pois usamos transação (**BOX 1**) no mesmo. Precisamos que os dados da OS e as peças que estão relacionadas ao mesmo sejam salvas no banco. Não podemos ter dados da Ordem sem peças ou peças sem uma Ordem.

BOX 1. Transação

Uma transação é uma sequência de operações em um sistema de gerência de banco de dados que são tratadas como um bloco único e indivisível (atômico) durante uma recuperação de falhas e também provê isolamento entre acessos concorrentes na mesma massa de dados.

Usamos transação, pois assim, garantimos que os dados sempre serão salvos com seus relacionamentos. Na **Listagem 1** temos o exemplo de utilização de uma transação usando dbExpress.

Temos uma função no DataModule que executa as Stored Procedures da OS e retorna o número da mesma que foi inserido. Passamos como parâmetro um ClientDataSet que é criado em memória na tela da Ordem de serviço.

Esse ClientDataSet possui as peças/serviços escolhidos pelo usuário. Iniciamos o código, configurando a variável da transação, usando o BeginTransaction (linha 9) e indicando seu nível de isolamento (**BOX 2**).

Após, dentro de um bloco try...except configuramos os parâmetros das Stored Procedures envolvidas na transação (linhas 13-15). O bloco try...except é de suma importância, pois qualquer erro, precisamos reverter os comandos executados.

Listagem 1. Exemplo de transação com dbExpress

```
01 function TDM.InserirOS(nCdOrcamento, nCdVeiculo: integer;
02 cdsItens: TClientDataSet; sDsObservacoes: string): integer;
02 var
03  TransDesc: TDBXTransaction;
04  nCdOrdem: integer;
05  i: integer;
06 begin
07
08  //inicia a transação
09  TransDesc := SysCar.BeginTransaction(TDBXIsolations.ReadCommitted);
10
11  try
12   //insere o orçamento
13   spOS.Params[0].AsInteger := nCdVeiculo;
14   spOS.Params[1].AsInteger := nCdOrcamento;
15   spOS.Params[2].AsString := sDsObservacoes;
16   spOS.ExecProc;
17
18   //armazena o valor inserido no banco
19   nCdOrdem := spOS.Params[3].AsInteger;
20
21   //percorre os itens e insere no banco
22   for i := 0 to cdsItens.RecordCount - 1 do
23   begin
24    //insere os itens
25    spOS_Itens.Params[0].AsInteger := nCdOrdem;
26    spOS_Itens.Params[1].AsInteger :=
27     cdsItens.FieldByName('nCdPeca').AsInteger;
28    spOS_Itens.Params[2].AsInteger :=
29     cdsItens.FieldByName('nQtde').AsInteger;
30    spOS_Itens.Params[3].AsCurrency :=
31     cdsItens.FieldByName('nVlPeca').AsCurrency;
32    spOS_Itens.ExecProc;
33
34    cdsItens.Next;
35   end;
36
37  except
38   //caso ocorra algum erro, da roolback
39   SysCar.RollbackIncompleteFreeAndNil(TransDesc);
40  end;
41
42  //comita as alterações no banco
43  SysCar.CommitFreeAndNil(TransDesc);
44
45  Result := nCdOrdem;
46 end;
```

BOX 2. Nível de isolamento

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações simultâneas. Os fenômenos não desejados são:

- dirty read (leitura suja) A transação lê dados escritos por uma transação simultânea não efetivada (uncommitted).
- nonrepeatable read (leitura que não pode ser repetida) A transação lê novamente dados lidos anteriormente, e descobre que os dados foram alterados por outra transação (que os efetivou após ter sido feita a leitura anterior).
- phantom read (leitura fantasma) A transação executa uma segunda vez uma consulta que retorna um conjunto de linhas que satisfazem uma determinada condição de procura, e descobre que o conjunto de linhas que satisfazem a condição é diferente por causa de outra transação efetivada recentemente.

Os quatro níveis de isolamento de transação, e seus comportamentos correspondentes, estão descritos na **Tabela 1**.

Nível de isolamento	Dirty read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

Tabela 1. Nível de isolamento

Dentro do except chamamos o RollbackIncompleteFreeAndNil (linha 39) para reverter os comandos executados e não inserir nada no banco de dados. Após o bloco, pois, caso nenhum erro ocorra, o fluxo do método não será executado no except, executamos o método CommitFreeAndNil que irá confirmar os comandos da Stored Procedure.

Por fim, retornamos o código da Ordem de Serviço inserida. Com isso, nosso método usando transação está completo. Caso haja dúvidas, o projeto dbExpress está disponível para download.

Migração para FireDAC

Agora vamos às modificações do projeto para usar o FireDAC. Vamos tentar mostrar um caminho mais suave, pois em aplicações reais não teremos apenas algumas telas e a utilização de poucos controles de dados, como vimos no projeto dbExpress.

É importante fazer um backup dos fontes antes da migração, se tiver uma ferramenta de versionamento, melhor. Crie uma versão para seu código, para que caso algum problema ocorra, seja fácil retornar ao ponto anterior. Vamos levar em conta que o FireDAC já está instalado no ambiente de desenvolvimento.

Nota

O FireDAC veio para substituir o dbExpress, mas o mesmo continuará a ser suportado, apenas não será mais atualizado.

Voltando ao projeto, precisamos modificar o controle de conexão com o banco de dados. No DataModule adicionamos um ADConnection. Com um duplo clique abrimos o editor de configuração de conexão (**Figura 5**).

O driver em uso é para SQL Server, mas note que temos muitas outras opções. Podemos usar o Test para verificar se a configuração está correta. Temos ainda um assistente para configurar a conexão com o banco de dados.

Após isso, alteramos a propriedade LoginPrompt para false e Connected para true, para testar a conexão com o banco.

Novos componentes de consulta

Com o FireDAC não precisamos usar um SQLDataSet e um DataSetProvider para as consultas (com o ClientDataSet). Basta usamos apenas um ADQuery para cada consulta.

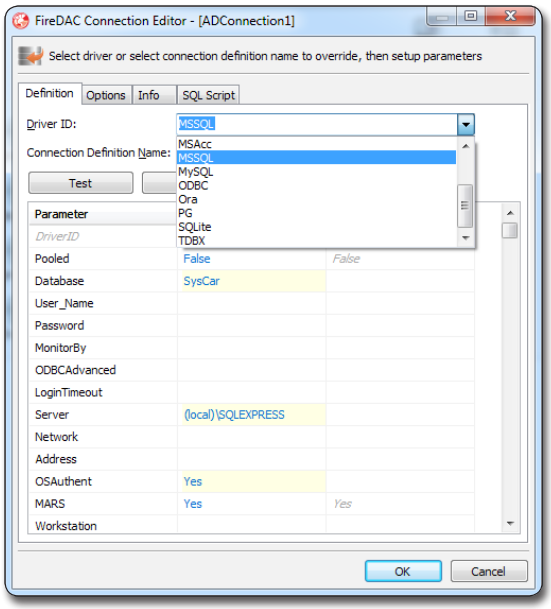

Figura 5. Criando uma conexão

Nota

Procure colocar o mesmo nome dos novos componentes para os antigos. Poderemos ter problemas com métodos, mas veremos isso, caso ocorra no decorrer do artigo.

No ADQuery aplicamos um duplo clique para abrir o editor de SQL. Podemos digitar o SQL da **Figura 6** e pressionar o botão Execute para realizar um simples teste.

Já temos uma facilidade com o FireDAC, pois podemos verificar em design-time se as consultas estão corretas, campos retornados etc. Faça o mesmo para cada ClientDataSet de consulta que temos no projeto.

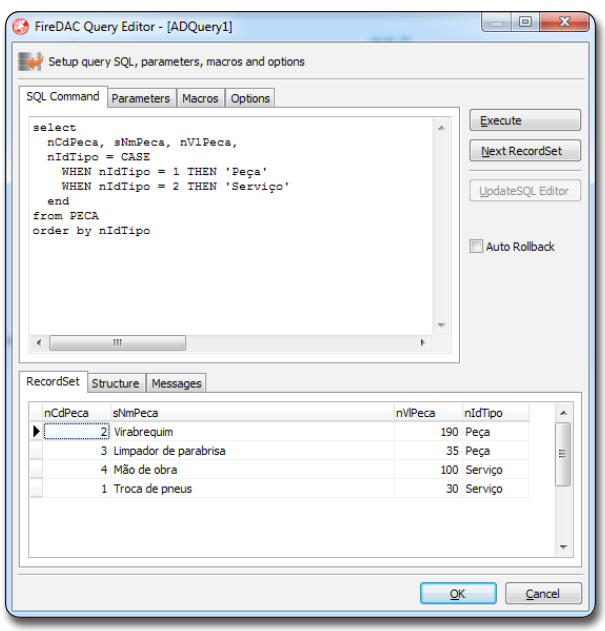

Figura 6. Editor de consultas do ADQuery

Para consultas onde temos parâmetros, no editor, na aba Parameters configuramos as informações do(s) parâmetro(s) da consulta, assim como podemos testar o retorno (**Figura 7**).

A aba Options possui muitas configurações como quantidade retornada pela consulta, lock etc. Veja na **Figura 8** as modificações já implementadas no DataModule.

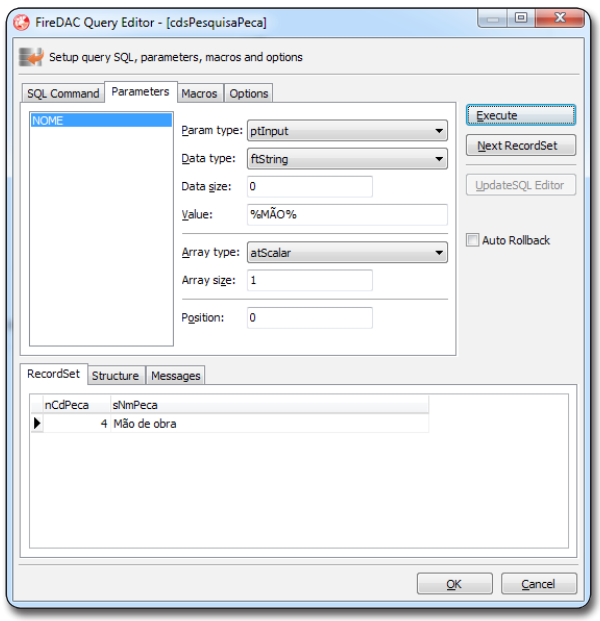

Figura 7. Configurando e testando os parâmetros do ADQuery

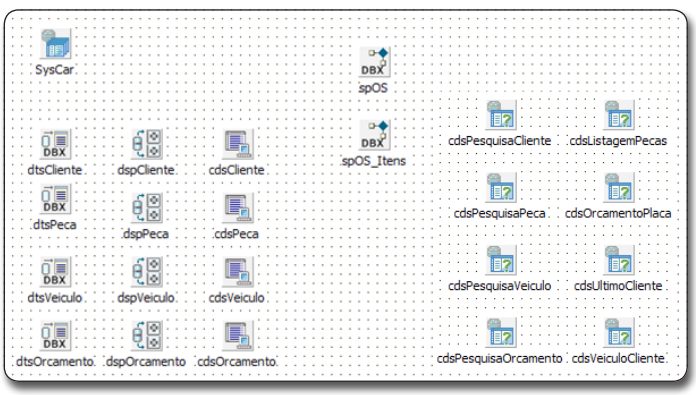

Figura 8. DataModule com modificações

Note que não precisamos indicar para cada ADQuery qual o seu componente de conexão (propriedade Connection). Ao adicionar o ADQuery no DataModule, ele procura pelo ADConnection e já configura a propriedade. Prático para não precisar configurar vários controles.

Cadastros

Para os cadastros vamos utilizar dois controles, ao invés do trio do dbExpress. Vamos usar um ADQuery para a consulta que mostraremos dos dados e um ADUpdateSQL onde teremos os comandos SQLs para manipular o cadastro (insert, update, delete).

Para quem usou o BDE, será bastante semelhante. Vamos configurar o cadastro de clientes como primeiro exemplo.

Adicionamos um ADQuery e um ADUpdateSQL. No ADQuery configuramos a propriedade UpdateObject com o ADUpdateSQL e configuramos a consulta e o parâmetro do cadastro.

Com um duplo clique no ADUpdateSQL é perguntado se desejamos pegar as informações das colunas da consulta para configurar. Ao escolher Yes (sim), será aberto um editor (**Figura 9**).

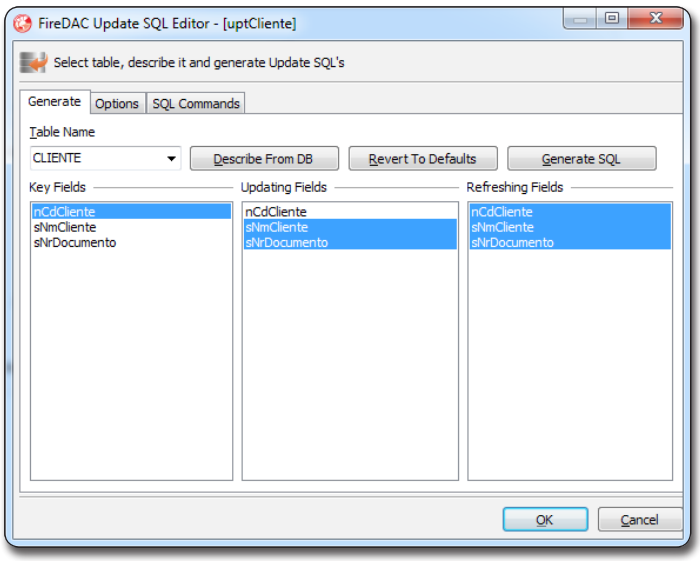

Figura 9. Editor do ADUpdateSQL

Vamos configurar o comportamento para os comandos de inserção, exclusão e atualização. Em Key Fields, selecionamos apenas a chave primária da consulta (nesse caso o nCdCliente). Em Updating Fields indicamos quais campos devem ser atualizados.

No caso de campos onde não precisamos atualizar (campos de consultas auxiliares, por exemplo), podemos deixar os mesmos desmarcados.

Nota

Esse caso é o mesmo que configuramos no pfInKey, pfInWhere e pfInUpdate da propriedade ProviderFlags do campo, no ClientDataSet.

Na aba Options temos configurações como habilitar os comandos (insert, update, delete), tipo de lock para a tabela, tipo de refresh dos dados, tipos de atualização etc. A aba SQL Commands apresenta os comandos SQLs criados para o cadastro. Para visualizar os comandos, clicamos no Generate SQL, na aba Generate. Os comandos gerados são vistos na **Figura 10**.

Após isso continuamos com os outros cadastros. Não usamos um ADTable para o cadastro, pois queremos continuar com a performance da aplicação onde nossos cadastros são parametrizados, retornando um registro por vez. Com o ADTable todos os registros seriam retornados sem necessidade. É possível notar no DataModule a diminuição de controles (**Figura 11**).

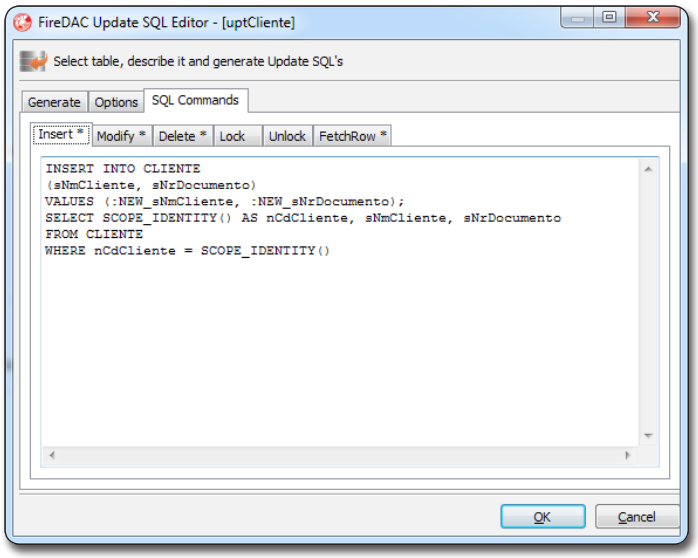

Figura 10. Configurando os comandos SQLs para o cadastro de cliente

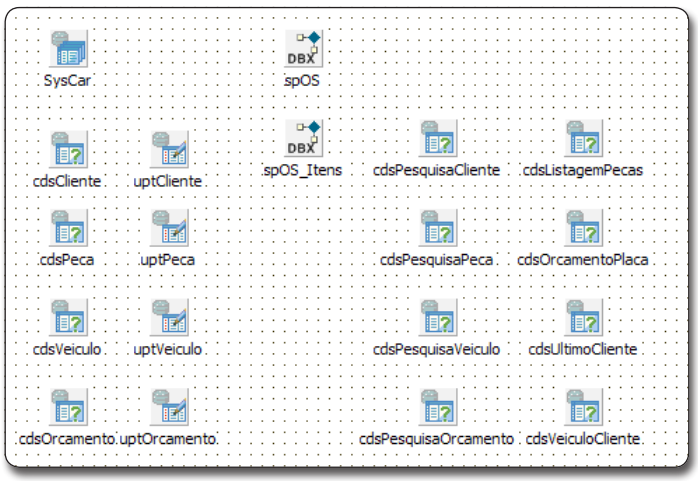

Figura 11. Quantidade de controles de dados diminui no DataModule

Stored Procedures

A utilização de Stored Procedures no FireDAC é semelhante ao dbExpress, temos um controle específico para isso. Vamos usar dois ADStoredProc configurando cada um para sua respectiva procedure no banco. Na propriedade Params, temos todos os parâmetros da procedure (**Figura 12**).

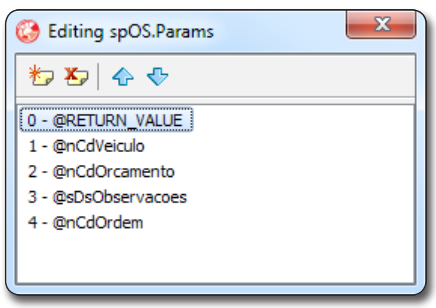

Figura 12. Parâmetros da Stored Procedure

O parâmetro sDsObservacoes se refere ao campo de mesmo nome na tabela OS. Ele é do tipo varchar(max). O FireDAC mapeia o mesmo com o DataType do tipo ftMemo. Assim, não podemos usar, por exemplo:

```
spOS.Params[3].AsString := sDsObservacoes;
```

Devemos usar:

```
spOS.Params[3].AsMemo := sDsObservacoes;
```

O mesmo vale para os parâmetros nQtdPeca e nVlUnitario da procedure de itens. O DataType é ftBCD. O método que salva a Ordem de serviço e seus itens foi mostrado na **Listagem 1**. Precisamos fazer algumas modificações.

A primeira, devemos adicionar um ADTransaction no DataModule. Após, devemos indicar a propriedade Transaction dos controles ADStoredProc para o novo controle. Na **Listagem 2** temos a última modificação, referente ao método para salvar OS e itens.

Listagem 2. Modificação do InserirOS

```
01 function TDM.InserirOS(nCdOrcamento, nCdVeiculo: integer;
02  cdsItens: TClientDataSet; sDsObservacoes: string): integer;
03 var
04  nCdOrdem: integer;
05  i: integer;
06 begin
07   //inicia a transação
08   transaction.StartTransaction;
09
10   try
11    //insere o orçamento
12    spOS.Params[1].AsInteger := nCdVeiculo;
13    spOS.Params[2].AsInteger := nCdOrcamento;
14    spOS.Params[3].AsMemo := sDsObservacoes;
15    spOS.ExecProc;
16
17    //armazena o valor inserido no banco
18    nCdOrdem := spOS.Params[4].AsInteger;
19
20    //percorre os itens e insere no banco
21    for i := 0 to cdsItens.RecordCount - 1 do
22    begin
23     //insere os itens
24     spOS_Itens.Params[1].AsInteger := nCdOrdem;
25     spOS_Itens.Params[2].AsInteger :=
26      cdsItens.FieldByName('nCdPeca').AsInteger;
27     spOS_Itens.Params[3].AsBCD :=
28      cdsItens.FieldByName('nQtde').AsCurrency;
29     spOS_Itens.Params[4].AsBCD :=
30      cdsItens.FieldByName('nVlPeca').AsCurrency;
31     spOS_Itens.ExecProc;
32
33     cdsItens.Next;
34    end;
35   except
36    //caso ocorra algum erro, da roolback
37    transaction.Rollback;
38   end;
39   //comita as alterações no banco
40   transaction.Commit;
41   Result := nCdOrdem;
42 end;
```

Veja que as modificações são simples. Não precisamos de uma variável e usamos os métodos do ADTransaction para iniciar a transação (linha 8), confirmar (linha 40) ou cancelar a mesma (linha 37).

Outras modificações

Precisamos modificar alguns códigos do projeto, pois fizemos casting para o ClientDataSet explicitamente. Precisamos mudar o código para fazer referência ao ADQuery. Os códigos a serem modificados estão no formulário base. Veja na **Listagem 3** as alterações.

Listagem 3. Alterações no formulário base de cadastro

```
//Método Pesquisar
procedure TfrmCadastro.Pesquisar;
begin
 {: pesquisa o registro }
 dsPesquisa.DataSet.Close;
 (dsPesquisa.DataSet as TADQuery).Params[0].AsString :=
  UpperCase('%' + edtPesquisa.Text + '%');
 dsPesquisa.DataSet.Open;
end;

//Evento DblClick do Grid
procedure TfrmCadastro.DBGrid1DblClick(Sender: TObject);
begin
 {: filtra os dados e mostra a aba de dados cadastrais }
 dsCadastro.DataSet.Close;
 (dsCadastro.DataSet as TADQuery).Params[0].AsInteger :=
   dsPesquisa.DataSet.Fields[0].AsInteger;
 ...
end;

//Método para ser usado após salvar e excluir
procedure TfrmCadastro.AtualizarCadastro;
begin
 dsCadastro.DataSet.Close;
 (dsCadastro.DataSet as TADQuery).Params[0].AsInteger := 0;
 dsCadastro.DataSet.Open;
end;
```

Usamos nesse projeto um componente para pesquisar registros, conforme comentamos anteriormente. Uma pesquisa auxiliar encapsulada em um controle de tela. Esse controle faz referência explícita ao ClientDataSet, portanto foi gerada uma nova versão do mesmo, referenciando o ADQuery.

Executando a aplicação

Agora estamos prontos para testar a aplicação que usa o FireDAC. Ao rodar a aplicação temos a mensagem de erro da **Figura 13**.

Essa mensagem ocorre por que precisamos adicionar na aplicação a unit ou o controle referente ao driver utilizado (nesse caso o SQL Server). Isso ocorre por que o FireDAC não usa DLL, tudo está no fonte da unidade, então precisamos adicionar um ADPhysMSSQLDriverLink ou colocar no uses do DataModule o "uADPhysMSSQL".

Outro erro que podemos ter é o da **Figura 14**.

Esse caso é igual, basta adicionar o ADGUIxWaitCursor ou no uses do DataModule o "uADGUIxFormsWait". Rode a aplicação, execute os cadastros, faça pesquisas, confirme, cancele etc.

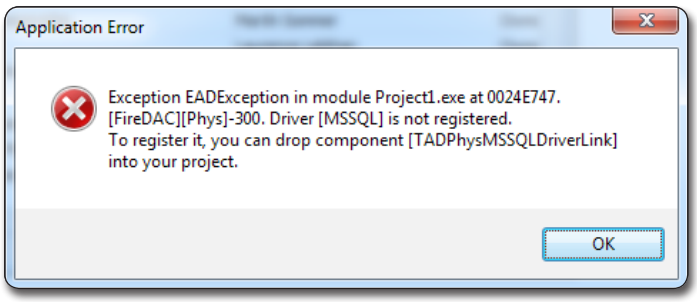

Figura 13. Mensagem de erro ao executar a aplicação

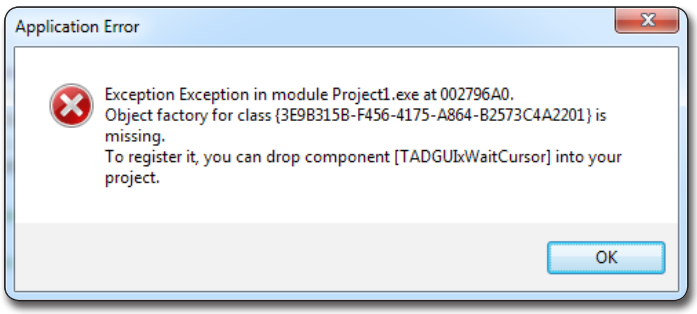

Figura 14. Mensagem de erro

Conclusão

Neste artigo foi apresentada uma maneira fácil para migrar seus projetos dbExpress para FireDAC. Claro, cada projeto tem suas características e peculiaridades. Analise bem seu projeto antes de fazer uma migração.

Tente achar todos os problemas que podem ocorrer nesse processo que não é fácil e de extrema complexidade. Caso sua análise indique que não vale a pena uma migração, comece a usar FireDAC em seus novos projetos.

Se sua aplicação é projetada em módulos e suas conexões são independentes, as facilidades aumentam, pois novos módulos podem ser desenvolvidos com FireDAC, ou a migração pode ser feita por módulos de maneira menos complexa.

Autor

Luciano Pimenta
luciano@lucianopimenta.com – www.lucianopimenta.com
Desenvolvedor Delphi/C# para aplicações Web com ASP.NET, Windows com Win32 e Windows Forms com .NET. Palestrante da 4ª edição da Borland Conference (BorCon) e da 1ª Delphi Conference.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

TMS Pack for FireMonkey

Pacote de componentes FireMonkey para desenvolvimento cross platform

TMS Pack for FireMonkey é um conjunto de componentes específicos para o framework cross-platform de aplicações da Embarcadero, o FireMonkey. Logo, cada componente presente no pacote é projetado para ser utilizado em interfaces de usuário de aplicações voltadas a esta tecnologia. Isso significa dizer que eles estarão presentes no desenvolvimento de aplicações Delphi para diferentes plataformas e dispositivos, vide PCs (Windows 32/64 bits e Mac OS X) e mais recentemente, tablets e smartphones (iOS). Por conseguinte, o presente artigo trará uma visão geral dos principais componentes presentes no pacote, abordando seus conceitos relacionados tanto no aspecto técnico quanto no aspecto prático.

Componentes

Uma vez que é instalado, o TMS Pack for FireMonkey adiciona à Tool Palette do IDE do Delphi duas novas categorias de componentes, denominadas TMS FMX UI e TMS FMX Instrumentation (**Figura 1**). Cada categoria engloba uma extensa quantidade de componentes, o que inviabiliza, por questões de espaço, uma abordagem totalmente completa do pacote. Dessa forma, aqui serão tratados apenas alguns deles, com a intenção de enfatizar aqueles considerados mais usuais.

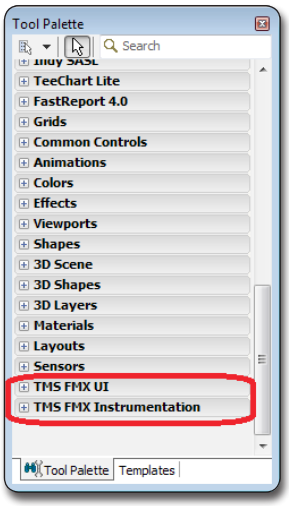

Figura 1. Categorias de componentes do TMS Pack for FireMonkey

Resumo DevMan

TMS Pack for FireMonkey:

O presente artigo tem como foco a apresentação de uma opção, dentre as várias disponíveis no mercado, de pacote de componentes de terceiros para FireMonkey. Especificamente será abordado o TMS Pack for FireMonkey, que é um conjunto de componentes provido pela empresa TMS Software, projetado para o uso no desenvolvimento cross-platform (multiplataforma) com FireMonkey no Delphi. Tal conhecimento é de suma importância na construção de aplicações mais robustas, uma vez que este pacote vem a complementar a gama de componentes disponíveis nativamente no IDE.

Em que situação o tema é útil:

Com o recente lançamento de sua versão XE4, o Delphi teve seu leque de recursos expandido com a adição do suporte ao desenvolvimento Mobile para a plataforma iOS. Isso veio a solidificar ainda mais o conceito cross-platform adotado pela ferramenta, em complemento ao suporte ao Win32, Win64 e Mac OS X já anteriormente existentes. Em meio a esta evolução o FireMonkey, em razão de sua característica multiplataforma, se apresenta naturalmente como o elemento chave que irá possibilitar a construção de aplicações para todas essas plataformas. Dito isto, torna-se essencial a busca por seu aprendizado mais aprofundado, tal como o conhecimento sobre opções de componentes de terceiros, que visam suprir algumas necessidades até então não atendidas pelos componentes nativos.

TTMSFMXPopup

Componente de pop-up cujo layout é projetado com base na janela pop-up nativa do iOS. Somente essa característica já o difere de qualquer elemento padrão similar, contudo, ele apresenta outras capacidades singulares. A começar pelo seu design totalmente personalizável, partindo de uma estrutura composta por cabeçalhos, rodapés, botões e conteúdos formatados em HTML. Além disso, o componente provê internamente a visualização de qualquer tipo de controle FireMonkey.

TTMSFMXRating

Componente simples de classificação, cuja estilização básica apresenta o indicativo de estrelas. Em uma dada aplicação, o mesmo poderia ser utilizado para indicar o nível de satisfação

do usuário perante alguma informação ou serviço, através do apontamento do número de estrelas.

TTMSFMXSearchEdit

Este componente nada mais é do que uma variação do TEdit clássico do FireMonkey, voltado ao contexto de buscas, tal qual seu nome sugere. Para isso são adicionados à sua estrutura visual dois botões, interpretados como Find e Clear, em alusão aos processos de pesquisar e apagar o conteúdo do controle. Em razão disso o componente conta ainda com eventos exclusivos (OnClickFind e OnClickClear) para a interação de ambos, assim como propriedades para o manuseio da visibilidade de cada um (ShowFind e ShowClear).

TTMSFMXSpeedButton

Componente que descende de TSpeedButton, incorporando a capacidade de exibir uma imagem em seu conteúdo. Além disso, TTMSFMXSpeedButton provê ainda uma propriedade denominada StaysPressed, que causa o efeito de manter o botão pressionado assim que é clicado. Aliado a essa funcionalidade, está outra propriedade de nome GroupName. Através dela, é possível se definir um nome para determinado agrupamento de botões do tipo TTMSFMXSpeedButton e fazer com que cada elemento atue, por exemplo, como um RadioButton. Na prática isto faz com que sempre apenas um botão do grupo fique pressionado.

TTMSFMXClock

Componente de relógio que opera basicamente em dois modos. Quanto "ativo" (propriedade Active = True), sua marcação de tempo se baseia no horário atuante na própria máquina em que está sendo rodado o aplicativo que o possui. Do contrário, a marcação de tempo toma como base o valor definido em sua propriedade ClockTime, tal como a seguir:

```
TMSFMXClock1.ClockTime := StrToTime('12:00');
```

TTMSFMXTableView

Controle para exibição de dados e informações diversas em diferentes modos, tais como no formato de lista ou de grupo (como no iOS). Cada item exibido é totalmente personalizável, o que acaba por contribuir na elaboração de layouts mais rebuscados e inovadores, tirando proveito de todo poderio de estilização que o próprio FireMonkey proporciona. Além disso, cada item pode conter uma parte relacionada à visualização de seus detalhes (DetailView), com suporte a qualquer controle FireMonkey para a exibição das informações. Adicionalmente, o componente provê ainda suporte integrado de busca, filtragem e edição de dados.

Além da parte de conteúdo, sua composição estrutural conta ainda com partes demarcatórias tradicionais como cabeçalho e rodapé.

TTMSFMXTileList

Componente que faz uso do conceito de Tiles (**BOX 1**), introduzido pelo Windows 8, o que acaba por traduzir seu objetivo pela definição de layouts neste novo estilo de design. Sua estrutura e seu aspecto visual são então compostos por uma coleção de Tiles totalmente personalizáveis. Como não poderia deixar de ser, o componente conta com recursos essenciais para o contexto, tais como navegação direta, multiseleção, busca e filtragem de dados, fluidez de rolagem, detalhamento por Tile e formatação HTML.

BOX 1. Tiles

O conceito de Tiles (Live Tiles) trazido pelo Windows 8 definiu um novo paradigma de interfaces de usuário ao utilizar grandes e coloridos blocos informativos associados às aplicações, de forma a servirem como um link para elas, exibindo informações mais recentes de status e atualizações. Em uma definição adicional, um Tile pode ser considerado a representação do aplicativo na tela inicial do Windows 8.

TTMSFMXGrid

TTMSFMXGrid é o principal (e único) componente de Grid presente no pacote TMS Pack. Apesar de sua esperada semelhança ao TStringGrid nativo do FireMonkey, conta com diversos recursos adicionais. Sua estrutura básica não foge do tradicional, sendo composta por uma matriz de células que dá forma ao seu aspecto de grade. Dentre emaranhados tipos de células, há aquelas consideradas "normais" - (TTMSFMXGridCell), e outras denominadas "fixas" (TTMSFMXFixedGridCell), que não permitem edição e não são movimentadas durante a rolagem, estando sempre visíveis (Ex: as células de títulos das colunas).

Propriedades

A começar por suas propriedades, alguns de seus diferenciais já podem ser notados. Como exemplo, ScrollMode possibilita se fazer a escolha entre dois tipos de rolagem para a grade: CellScrolling e PixelScrolling. O primeiro determina um deslocamento baseado em linhas e colunas inteiras, mantendo assim as informações contidas nas células idealmente sempre visíveis. Já o segundo tipo se baseia na largura e altura total das células, provendo um deslocamento mais preciso, e que durante a navegação causa o efeito de células parcialmente visíveis.

Já com a propriedade SelectionMode é possível se definir o tipo de seleção que será permitida no Grid. No total são dez opções diferentes:

- smSingleCell: permite a seleção de apenas uma única célula;
- smSingleRow: permite a seleção de uma linha completa;
- smSingleColumn: permite a seleção de uma coluna completa;
- smNone: sem seleção de célula;
- smCellRange: permite a seleção de várias células subsequentes, determinando um intervalo;
- smRowRange: permite a seleção de várias linhas subsequentes;
- smColumnRange: permite a seleção de várias colunas subsequentes;
- smDisjunctCell: permite a seleção de várias células aleatórias;
- smDisjunctRow: permite a seleção de várias linhas aleatórias;
- smDisjunctColumn: permite a seleção de várias colunas aleatórias.

Adicionalmente, em Options ficam localizadas algumas opções de configuração do componente. Apenas para citar duas delas, em Band é possível se determinar a quantidade de linhas coloridas alternativas, que irão causar, por exemplo, um efeito visual "zebrado" nas linhas do Grid. Já em Clipboard é onde se habilita ou não os tradicionais atalhos de copiar e colar – Ctrl+C, Ctrl+X e Ctrl+V – para serem usados no componente em execução.

Tipos específicos de células

Com o TTMSFMXGrid é possível ainda se definir, de forma dinâmica, o tipo da classe de determinada célula. Para tal, além de seus tipos mais básicos já citados – TTMSFMXGridCell e TTMSFMXFixedGridCell - o componente oferece uma série de outras classes pré-definidas de células mais específicas, todas herdando da classe base TTMSFMXGridCell. Como exemplo, TTMSFMXRadioGridCell que é a classe para uma célula com capacidade de exibir internamente um controle RadioButton. Da mesma forma, além desta, algumas outras classes presentes são: TTMSFMXCheckGridCell (para exibição de um CheckBox), TTMSFMXButtonGridCell (para exibição de um Button), TTMSFMXProgressGridCell (para exibição de um ProgressBar), TTMSFMXCommentGridCell (para exibição de um pop-up informativo) e TTMSFMXBitmapGridCell (para exibição de uma imagem).

Merge e Split de células

O TTMSFMXGrid apresenta também o recurso de mesclar (Merge) e dividir (Split) células, provido por dois meios: programação (via código) ou teclado (via teclas de atalho). Para o primeiro, são providos três métodos, cada qual com sua finalidade, de acordo com seu nome – MergeCells, MergeSelection e SplitCell. Já para os atalhos de teclado, são utilizadas as combinações Ctrl + M para Merge e Ctrl + S para Split, a partir de uma seleção de células. Essa funcionalidade é dependente da "ativação" da propriedade AllowCellMergeShortCuts do componente, tal como o comando a seguir:

```
TMSFMXGrid1.Options.Keyboard.AllowCellMergeShortCut := True;
```

Cálculos

O Grid trazido pelo TMS Pack for FireMonkey conta ainda com métodos para a realização de cálculos sobre suas linhas e colunas. Na prática eles são disponibilizados como funções (Functions) da linguagem, tais como ColumnSum, que calcula a soma dos valores das células de determinada coluna, e ColumnAvg, que calcula o valor médio das mesmas.

Importação e Exportação de dados

O TTMSFMXGrid também contempla recursos para importação e exportação de dados em diversos formatos diferentes: XML, XLS, HTML, RTF, CSV, entre outros.

Ordenação

O componente conta ainda com recursos integrados para classificação/ordenação de dados. Eles são disponibilizados em três modos de operar diferentes:

- None: sem ordenação;
- Normal: habilita a ordenação de dados de forma ascendente/descendente em uma dada coluna;
- Indexed: habilita a ordenação de dados de forma ascendente/descendente em várias colunas.

Programaticamente, a habilitação do modo indexado poderia ser feita da seguinte forma:

```
TMSFMXGrid1.Options.Sorting.Mode := gsmIndexed;
```

Obviamente, o componente TTMSFMXGrid possui ainda outra grande quantidade de funcionalidades, mas que demandaria um artigo todo dedicado. Dessa forma, aqui foram expostas somente algumas delas, a fim de se fazer uma apresentação inicial do componente.

TTMSFMXNavBar

TTMSFMXNavBar é um componente visual do tipo container, ou seja, uma espécie de controle que serve como recipiente para

outros componentes. Neste caso, tais componentes são essencialmente objetos do tipo TTMSFMXNavBarPanel que, conforme sua nomenclatura sugere, são elementos similares ao TPanel nativo. Isso significa dizer que cada painel deste servirá também de container para outros elementos, podendo ser qualquer controle FireMonkey (FMX).

A adição destes controles é que irá definir todo o aspecto da interface de usuário que está sendo montada. É possível ainda se definir para cada painel, uma série de seções lógicas que, internamente são representadas por objetos do tipo TTMSFMXNavBarPanelSection. Na medida em que um painel é adicionado, uma lista contendo todos os painéis vai sendo montada, com recursos de navegação e recolhimento. Numa exemplificação visual, a **Figura 2** mostra a utilização de um TTMSFMXNavBar em um projeto FireMonkey Desktop Application. Aqui é possível notar a presença de três painéis, onde cada painel representa uma revista da editora DevMedia (neste caso Clube Delphi, .Net Magazine e Java Magazine). Para o painel ativo, "Edição 100" é o texto dado à seção interna definida.

Figura 2. TTMSFMXNavBar em execução

TTMSFMXEdit

Numa definição simples e popular, o TTMSFMXEdit pode ser considerado um TEdit "turbinado". Isto porque este componente apresenta uma série de recursos adicionais, inexistentes em seu "coirmão" nativo, tais como pesquisas do tipo Lookup e recurso de auto completar. Na prática, detalhes como estes contribuem para o aumento da experiência do usuário, tornando a aplicação ainda mais atrativa e fácil de usar.

Lookup

O recurso de Lookup exibe uma lista de opções com base nos caracteres que estão sendo digitados pelo usuário. Todas as configurações relacionadas ao recurso ficam concentradas na propriedade Lookup, que contempla um conjunto de subpropriedades listadas no Object Inspector. Além disso, é possível ainda realizar estas mesmas definições via código, tal como mostra a **Listagem 1**.

Neste trecho de código apresentado é possível notar o envolvimento básico de quatro propriedades diferentes do TTMSFMXEdit, relacionadas a Lookup:

- Enabled: ativa o recurso no componente (linha 01);
- DisplayCount: determina o número de itens opcionais que irão aparecer na lista no momento da digitação (linha 02);
- NumChars: determina o número de caracteres que deverão ser digitados para que a lista seja mostrada (linha 03);
- DisplayList: representa a lista de itens que serão exibidos (linhas 04 a 07);

O resultado do código da **Listagem 1**, na prática, é demonstrado na **Figura 3**.

Listagem 1. TTMSFMXEdit – Configuração do recurso de Lookup

```
01 TMSFMXEdit1.Lookup.Enabled := True;
02 TMSFMXEdit1.Lookup.DisplayCount := 4;
03 TMSFMXEdit1.Lookup.NumChars := 1;
04 TMSFMXEdit1.Lookup.DisplayList.Add('Revista Clube Delphi');
05 TMSFMXEdit1.Lookup.DisplayList.Add('Revista Net Magazine');
06 TMSFMXEdit1.Lookup.DisplayList.Add('Revista Java Magazine');
07 TMSFMXEdit1.Lookup.DisplayList.Add('Revista SQL Magazine');
```

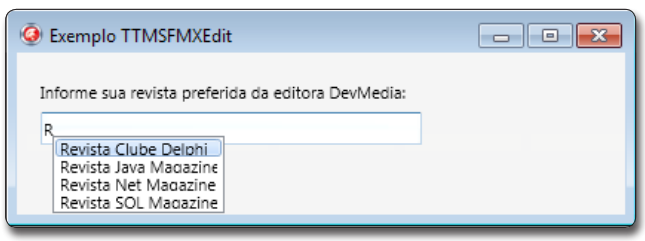

Figura 3. TTMSFMXEdit – Recurso de Lookup

Auto Complete

O recurso de auto completar (Auto Complete) do TTMSFMXEdit se aproveita da mesma propriedade relacionada à lista de itens do Lookup para efetivar a ação desejada. Isso significa dizer que é a partir dos itens definidos em DisplayList que o componente irá buscar a ocorrência mais próxima para auto completar o texto que está sendo digitado. A **Listagem 2** exibe o código necessário utilizado no exemplo da **Figura 4**. Conforme pode ser visto, a "ativação" do recurso no componente se dá única e exclusivamente pela sua propriedade AutoComplete.

Listagem 2. TTMSFMXEdit – Configuração do recurso de auto completar

```
01 TMSFMXEdit1.AutoComplete := True;
02 TMSFMXEdit1.Lookup.DisplayList.Add('Clube Delphi');
03 TMSFMXEdit1.Lookup.DisplayList.Add('Net Magazine');
04 TMSFMXEdit1.Lookup.DisplayList.Add('Java Magazine');
05 TMSFMXEdit1.Lookup.DisplayList.Add('SQL Magazine');
```

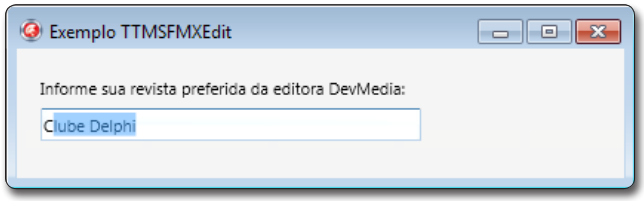

Figura 4. TTMSFMXEdit – Recurso de Auto Complete

TTMSFMXEditBtn

No popular, da mesma forma que o TTMSFMXEdit, o componente TTMSFMXEditBtn pode ser considerado uma evolução do TEdit tradicional. Todavia, no aspecto técnico ele vai além, contemplando todas as características de TTMSFMXEdit, uma vez que o estende com o adicional de um botão interno de edição. Uma vez acionado, este botão pode então invocar em modo pop-up um controle anexado ao componente, associado através de sua propriedade PopupControl. Esta por sua vez é capaz de receber qualquer controle do tipo proveniente de TControl. A **Figura 5** mostra o componente associado a um simples TListBox, tal como o código a seguir:

```
TMSFMXEditBtn1.PopupControl := ListBox1;
```

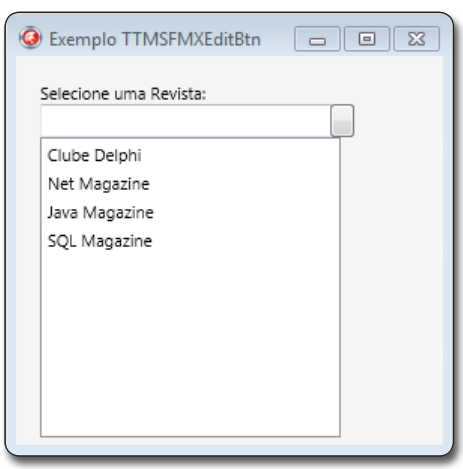

Figura 5. TTMSFMXEditBtn associado a um controle TListBox

TTMSFMX7SegLED

TTMSFMX7SegLED é um mostrador de valores numéricos, cuja representação de cada dígito se dá a partir de um elemento de sete segmentos de LED. Toda a configuração de valor e exibição do componente gira em torno de três de suas propriedades:

- Value: define o valor (o número) a ser exibido pelo controle;
- Digits: determina a quantidade de dígitos para a parte inteira (antes do ponto decimal) do número a ser exibido;
- Decimals: determina a quantidade de dígitos para parte decimal do número a ser exibido.

Numa exemplificação simples, a exibição do número 10.11 tem sua implementação demonstrada na **Listagem 3**. O resultado pode ser visto na **Figura 6**.

Listagem 3. Representação do número 10.11

```
01 TMSFMX7SegLED1.Digits := 2;
02 TMSFMX7SegLED1.Decimals := 2;
03 TMSFMX7SegLED1.Value := 10.11;
```

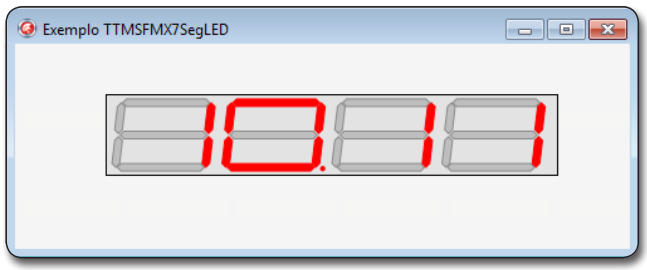

Figura 6. Componente TTMSFMX7SegLED em execução

Como pode-se notar, a **Listagem 3** ilustra através da propriedade Digits a quantidade de dígitos a serem exibidos (linha 01), quantidade de casas decimais (linha 02) e o valor a ser exibido, por meio da propriedade Value (linha 03).

TTMSFMXCompass

TTMSFMXCompass é um simples componente visual (controle) representativo de uma bússola. Em sua composição estética há a presença de uma agulha principal apontada para o Norte e outras oito agulhas subjacentes para cada direção do vento. Em termos práticos, RotationAngle é o nome da propriedade presente no componente que deverá ser manipulada para se realizar mudanças no que diz respeito a orientação da bússola. A **Figura 7** mostra o componente em uma aplicação FireMonkey em execução, cujo valor para a propriedade RotationAngle está definido como 0 (valor padrão).

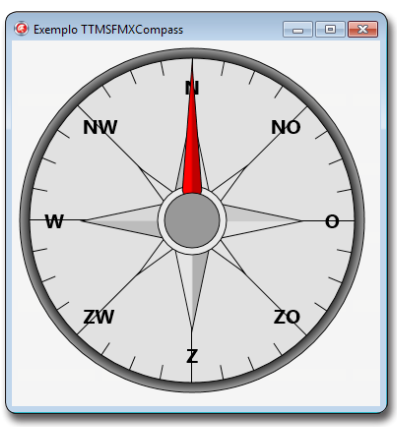

Figura 7. Componente TTMSFMXCompass em execução

TTMSFMXCalendar

TTMSFMXCalendar é um controle representativo de calendário, similar ao TCalendar nativo da biblioteca do FireMonkey, mas com o adicional de uma série de recursos extras.

Dentre estes, os destaques ficam por conta de sua capacidade de multiseleção de datas e seu trabalho com indicação de eventos em dias específicos.

Multiseleção

Para habilitar ou não seu recurso de multiseleção, o componente apresenta uma propriedade booleana denominada MultiSelect. Uma vez habilitado, o recurso possibilita a seleção de várias datas distintas ou mesmo um intervalo entre datas. A **Figura 8** mostra o comportamento do componente em execução, onde várias datas foram selecionadas, demarcando assim um intervalo entre elas. Na prática, esta seleção é determinada pelo simples arrastar do mouse com seu botão esquerdo pressionado sobre os dias desejados, ou segurando-se a tecla Shift em combinação com as teclas direcionais (setas) do teclado.

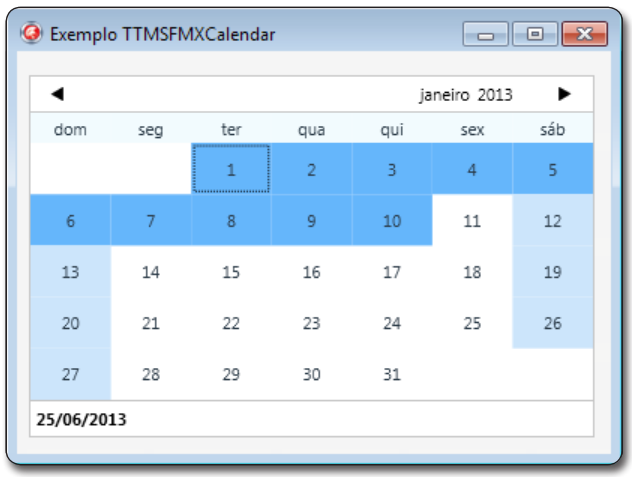

Figura 8. Componente TTMSFMXCalendar com multiseleção

Adicionalmente a multiseleção aleatória de datas é provida pela "ativação" de outra propriedade, denominada DisjunctSelect, em parceria com a atual MultiSelect. Com a aplicação em execução, segurando-se a tecla Ctrl, a seleção de datas distintas e aleatórias se dá pelo pressionamento da barra de espaço ou clique do mouse, conforme mostra a **Figura 9**. Por fim, para se recuperar as datas selecionadas, o TTMSFMXCalendar provê uma propriedade específica denominada SelectedDays, que representa a coleção (TDisjunctDateArray) de datas selecionadas. Em termos de código, com base no exemplo da figura anterior, um simples loop pelo Array já seria o suficiente para obter os valores contidos na coleção, tal como mostrado a seguir na **Listagem 4.**

No código da **Listagem 4** podemos observar primeiramente a criação de um StringList (linha 06). Na linha 08 é realizado um laço contado a partir da quantidade de dias selecionados (contidos dentro da propriedade SelectedDays, conforme mencionado). A seguir, na linha 09, é realizada a conversão dos valores do tipo Data para Texto, adicionando-os ao StringList. Por fim, na linha 10 é exibido o conteúdo do StringList, tendo sua destruição realizada na linha 12. O resultado desta listagem pode ser observado na **Figura 10.**

Listagem 4. Utilizando um loop para ler as datas selecionadas

```
01 var
02  I: Integer;
03  SL: TStringList;
04 begin
05  try
06   SL := TStringList.Create;
07   SL.Add('Datas selecionadas:');
08   for I := 0 to Pred(TMSFMXCalendar1.SelectedDays.Count) do
09    SL.Add(FormatDateTime('dd/MM/yyyy',
     TMSFMXCalendar1.SelectedDays[I].Date));
10   ShowMessage(SL.Text);
11  finally
12   SL.Free;
13  end;
14 end;
```

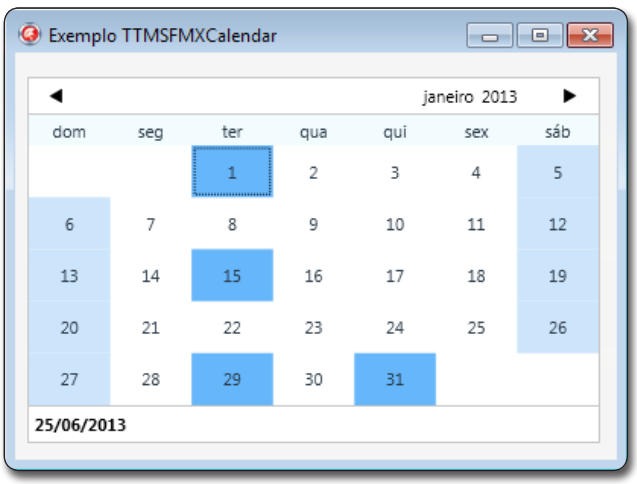

Figura 9. Componente TTMSFMXCalendar com multiseleção de datas aleatórias

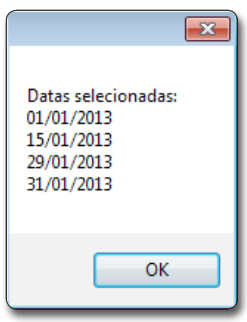

Figura 10. Mensagem mostrando as datas selecionadas em TTMSFMXCalendar

Eventos

Conforme mencionado, outra peculiaridade do TTMSFMXCalendar é seu trabalho com eventos. Para isso, o componente apresenta uma propriedade chamada Events, que então armazena uma coleção de objetos TTMSFMXCalendarEvent, onde cada qual representa um evento configurado. Neste ponto, a cada evento são atribuídas quatro informações: uma cor (Color), uma forma indicativa (Kind), um texto representativo (Text) e a data de exibição do evento (Date). Todos estes detalhes podem ser vistos na **Listagem 5**, que mostra um trecho de código que adiciona um novo evento ao componente, configurado para o dia 01/01/2013.

Observe na **Listagem 5** que um evento é declarado assim como um objeto qualquer (linha 02). Na linha 04 ele é instanciado, e, a seguir, é informado um modo de marcação em formato de elipse (linha 05). A data é atribuída e por último o texto (linhas 06 e 07). O resultado é ilustrado na **Figura 11**.

Listagem 5. TTMSFMXCalendar – Adicionando um evento ao componente

```
01 var
02   MeuEvento: TTMSFMXCalendarEvent;
03 begin
04   MeuEvento := TMSFMXCalendar1.Events.Add;
05   MeuEvento.Kind := ekEllipse;
06   MeuEvento.Date := StrToDate('01/01/2013');
07   MeuEvento.Text := 'Primeiro dia do ano!';
```

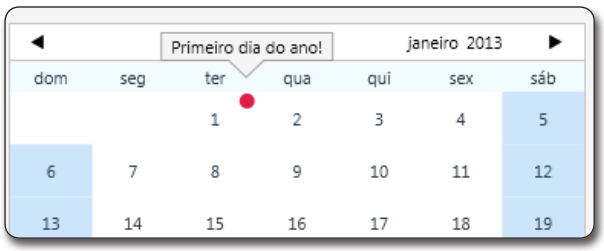

Figura 11. Evento sendo mostrado no componente em execução

TTMSFMXMemo

TTMSFMXMemo é um controle Memo, similar ao TMemo do FireMonkey, mas com uma porção de recursos adicionais, tais como autocorreção, preenchimento automático, marcadores e destaque configurável de sintaxe. Anexado a este último está também seu suporte a linguagens de programação, que envolve uma ampla gama de opções que incluem Pascal, Visual Basic, C#, SQL, HTML, CSS, JavaScript e XML. Para cada uma delas é então disponibilizado um componente específico, tratado como Styler, que se liga ao componente TTMSFMXMemo por meio da propriedade SyntaxStyles, atuando como um "estilizador" de sintaxe para o conteúdo do tipo de arquivo relacionado à linguagem. Sendo assim, como exemplo, TTMSFMXMemoHTMLStyler é o componente direcionado a HTML, TTMSFMXMemoCSharpStyler para C#, TTMSFMXMemoSQLMemoStyler para SQL, e assim por diante. Para a comunidade Delphi, um atrativo maior talvez se concentre no Styler para a linguagem Pascal (TTMSFMXMemoPascalStyler). Em vista disso, podemos ver seu uso na prática, a fim de se explorar algumas das propriedades presentes no componente, bem como o aspecto visual obtido no realce de sintaxe apresentado no Memo.

AutoCompletion

Em TTMSFMXMemoPascalStyler, sua propriedade AutoCompletion, do tipo TStringList, refere-se ao recurso de auto completar

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DevMedia

de código. Sendo assim, é nesta lista que deverão estar armazenadas as strings que aparecerão como sugestão no momento da invocação do recurso, por meio do atalho Ctrl + Barra. Por padrão, o componente já traz duas strings inseridas para as instruções ShowMessage e MessageDlg. Num demonstrativo simples de codificação, a adição de novas strings poderia ser feita conforme ilustra a **Listagem 6**.

Na **Listagem 6** podemos notar que pelo fato dos métodos Write e Writeln no Pascal exigirem um parâmetro, foram utilizados parênteses em suas strings. Isso faz com que no momento do autocompletar, o cursor já seja devidamente posicionado entre os parênteses para a digitação dos parâmetros.

Listagem 6. Adição de strings ao Memo

```
01 TMSFMXMemoPascalStyler1.AutoCompletion.Add('Write()');
02 TMSFMXMemoPascalStyler1.AutoCompletion.Add('Writeln()');
03 TMSFMXMemoPascalStyler1.AutoCompletion.Add('Read');
04 TMSFMXMemoPascalStyler1.AutoCompletion.Add('Readln');
```

BlockStart e BlockEnd

BlockStart e BlockEnd formam um casal de propriedades que, conforme seus nomes sugerem, se referem às indicações de início e fim de blocos de código da linguagem, armazenando os textos (strings) relacionados. No caso do Pascal, o único texto indicativo de fim é o "end", enquanto que para o indicativo de início podem ser vários, tais com "begin", "try", entre outros. Para facilitar, o componente já traz essas propriedades previamente preenchidas, conforme mostrado na **Listagem 7**.

Listagem 7. Preenchimento padrão

```
01 object TMSFMXMemoPascalStyler1: TTMSFMXMemoPascalStyler
...
02   BlockStart = 'begin,try,case,class,record'
03   BlockEnd = 'end'
...
04 end
```

CommentStyle e MultiComment

CommentStyle é uma propriedade do tipo TCharStyle que envolve um conjunto de valores que irão determinar o estilo a ser utilizado pelos textos de comentários presentes no código. Sendo assim, sua manipulação envolve configurações como cor de fundo, cor do texto e estilo da fonte (negrito, itálico, etc.). O código a seguir mostra a definição do estilo da fonte como itálico.

```
TMSFMXMemoPascalStyler1.CommentStyle.Style := [TFontStyle.fsItalic];
```

De forma conjunta à propriedade CommentStyle, estão outras três também relacionadas a comentários. Elas são apresentadas como LineComment, MultiCommentLeft e MultiCommentRight, e tem como função armazenar os caracteres que identificam o comentário de uma única linha e o de início e fim de um comentário de múltiplas linhas, respectivamente. Seguindo os padrões utilizados no Pascal, o componente já apresenta valores default para cada uma delas (**Listagem 8**).

Listagem 8. Valores padrão para comentários

```
01 object TMSFMXMemoPascalStyler1: TTMSFMXMemoPascalStyler
...
02   LineComment = '//'
03   MultiCommentLeft = '{'
04  MultiCommentRight = '}'
...
05 end
```

AllStyles

Esta propriedade provê um editor na forma de lista, que corresponde a uma coleção de TElementStyles. Cada item desta coleção trata da estilização de determinado elemento presente na linguagem. Por padrão, o componente apresenta cinco elementos TElementStyles já pré-definidos e devidamente configurados, mediante o padrão do Pascal. Como forma de apresentação, a seguir são expostos cada um deles.

- Pascal Standard Default: engloba as definições das palavras-chave (keywords) do Delphi/Pascal;
- Simple Quote: contém as definições de aspas simples;
- Double Quote: contém as definições de aspas duplas;
- Symbols Delimiters: envolve as definições de diferentes símbolos delimitadores;
- Multi line Comment: envolve as definições de comentário de múltiplas linhas;

Cada TElementStyles contempla outra gama de propriedades. Destaque para a propriedade Info, que mantém a denominação dada ao elemento, e StyleType, que demarca o tipo de estilo que está sendo definido: stKeyWord (para uso de palavras-chave), stBracket (para uso de descrições de apoio), stSymbol (para uso de símbolos) e stComment (para uso de comentários). Especificamente para o StyleType stKeyWord, sua escolha é automaticamente combinada à utilização da propriedade KeyWords, do tipo TStringList. Nela é onde estarão listadas todas as palavras-chave relacionadas ao contexto. Dentre os elementos TElementStyles trazidos por default pelo componente, Pascal Standard Default é o que possui o StyleType marcado como stKeyWord. Logo, ele já traz consigo uma lista de palavras-chave usuais no Pascal, conforme pode ser visto na **Listagem 9**.

Da mesma forma, Symbols Delimiters, que apresenta o StyleType marcado como stSymbol, traz em sua propriedade Symbols a lista de símbolos usuais da linguagem, tais como ( ), { }, [ ], =, +, -, #13 e #10. Por fim, para uma melhor visualização inicial dos recursos mencionados até aqui e, principalmente o realce de sintaxe proporcionado pelo componente, a **Figura 12** mostra o TTMSFMXMemo em ação conjunta com TTMSFMXMemoPascalStyler, numa codificação utilizando a linguagem Delphi.

A lista completa de componentes que compõem o pacote, bem como a documentação detalhada sobre cada elemento, incluindo diversos exemplos de utilização podem ser encontrados no próprio site do fabricante, disponível na seção **Links**.

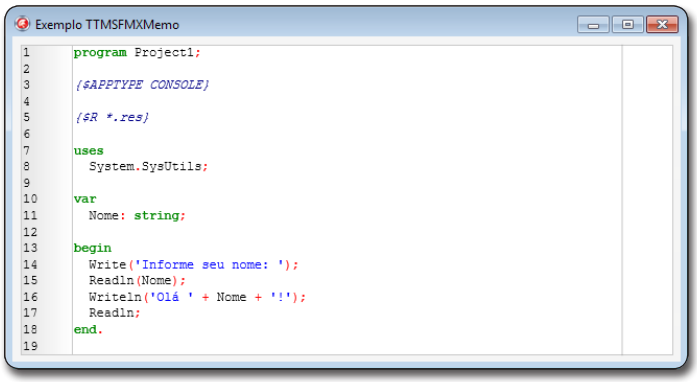

Figura 12. TTMSFMXMemo em execução

Listagem 9. Palavras reservadas para o Pascal

```
01 object TMSFMXMemoPascalStyler1: TTMSFMXMemoPascalStyler
...
02  AllStyles = <
03    item
04      Info = 'Pascal Standard Default'
05      KeyWords.Strings = (
06      'UNIT'
07      'PROGRAM'
08      'PRIVATE'
09      'PUBLIC'
10      'PUBLISHED'
11      'PROTECTED'
12      'PROPERTY'
13      'VAR'
14      'BEGIN'
15      'WITH'
16      'END'
17      'FOR'
18      'TO'
19      'DO'
20      'NOT'
21      'IF'
22      'THEN'
23      'ELSE'
24      'TYPE'
25      'WHILE'
26      'REPEAT'
27      ...)
28    end
29 ...
30 end
```

Conclusão

Em meio às recentes novidades incorporadas ao Delphi, o FireMonkey tem se estabelecido cada vez mais como sendo a tecnologia de futuro para a construção de aplicações Delphi. Isso é justificado principalmente pela sua característica cross-platform que o credencia para ser utilizado em projetos de desenvolvimento para Windows (32 e 64 bits), Mac OS X e iOS, atendendo a uma gama de diferentes dispositivos (PCs, Smartphones e Tablets).

Como consequência natural, a procura pelo estudo e maior aprofundamento de conhecimento sobre o FireMonkey tende a crescer gradualmente em meio à comunidade de desenvolvedores Delphi. Atualmente, isto já é notório em fóruns e comunidades espalhadas pela rede. Diante deste cenário, o presente artigo abordou um tema que também tende a crescer com o passar do tempo, que é o de componentes de terceiros para FireMonkey. Obviamente, as opções disponíveis no mercado atualmente são extremamente restritas se comparado à VCL. Todavia, estas mesmas empresas que hoje só oferecem pacotes de componentes para VCL, também já estão de olhos abertos para o mercado em expansão do FireMonkey.

Agradeço a todos novamente e até a próxima.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

Delphi XE4 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE4 – Página oficial do produto
http://www.embarcadero.com/products/delphi

TMS Pack for FireMonkey - Página oficial do produto
http://www.tmssoftware.com/site/tmsfmxpack.asp

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Desvendando Métodos Anônimos

Conheça a fundo este recurso pouco utilizado

Programadores Delphi na sua maioria estão acostumados a desenvolver sempre da forma mais RAD (Desenvolvimento Rápido de Aplicações) possível. A ferramenta nos permite em pouco tempo projetar sistemas e disponibilizar aos clientes em ambiente de produção. Porém esta não é a melhor forma de desenvolvimento, como já sabemos. Isto também não quer dizer que devemos instanciar todos os Forms, Buttons, Edits e demais componentes "na unha", para depois liberarmos todos manualmente. Devemos utilizar o modo RAD do Delphi para o desenvolvimento de interfaces (GUI). No entanto, para tarefas como efetuar conexões com banco de dados, escrever regras de negócios e rotinas de cálculo, por exemplo, deve-se usar todos os recursos que a Delphi Language nos disponibiliza, tornando o código mais limpo, facilmente expansível, além de contribuir com a facilidade de manutenção.

Métodos anônimos são um dos recursos da Delphi Language presentes a partir da versão 2009 do Delphi. Historicamente falando, esta versão trouxe diversas e importantes melhorias à Linguagem, trazendo consigo a nova RTTI recheada de melhorias, independência do DataSnap Framework da tecnologia COM+, inclusão dos Generics (**BOX 1**) e Unicode (**BOX 2**), por exemplo. Este recurso apesar de não ser tão recente, ainda é pouco utilizado pelos desenvolvedores.

Como o próprio nome sugere, os métodos anônimos são procedimentos e funções que não tem um nome associado. Podemos associar blocos de código a uma variável ou usarmos como parâmetro para um método. Também há como executar métodos através de chamadas sem nome, ou seja, podemos ter vários métodos sem nome e executá-los da melhor forma, cada um de acordo com a necessidade. É um recurso semelhante aos Delegates (**BOX 3**) e Closures (**BOX 4**) da plataforma .NET, além de serem usados por algumas outras linguagens de programação, como o JavaScript por exemplo.

Resumo DevMan

Desvendando Métodos Anônimos:

Neste artigo veremos o conceito de métodos anônimos, como fazer as declarações de tipo de método. Será feita uma revisão ao uso de variáveis, ponteiros para procedimentos, funções e métodos de objetos que geralmente são empregados em eventos.

Abordaremos algumas situações onde podemos empregar este recurso, como em DataSets, StringLists, gerenciamento de transações a nível de banco de dados, componentes visuais, Threads entre outros.

Em todos estes exemplos será realizado um comparativo do modo tradicional de implementação com o uso de métodos anônimos. Facilmente perceberemos o quanto este recurso da linguagem torna nosso código mais limpo, seguro e de fácil manutenção.

Em que situação o tema é útil:

É um recurso útil em situações de refatoração de código, minimizando códigos redundantes, demonstrando métodos que podem realizar procedimentos dinâmicos de acordo com a necessidade. Vale comentar que é aplicável no desenvolvimento de todos os tipos de aplicações Delphi, tanto em ambiente Client-Server quanto Multicamadas e Web.

No início, pode-se achar um pouco estranha sua utilização, mas ao compreender a sintaxe, os métodos anônimos são de extrema utilidade em várias situações, principalmente onde há iteração, como em listas de objetos, Strings, Arrays e DataSets. Veremos nos exemplos práticos como aplicar o recurso nestas situações.

Declaração

Métodos Anônimos podem ser definidos tanto como Procedures quanto Functions, além de poder possuir qualquer número de parâmetros ou nenhum. Podemos usar os já existentes na RTL do Delphi ou criarmos os nossos personalizados. Os tipos de métodos anônimos são declarados como referências para métodos (procedures ou functions). É possível atribuir o método anônimo a variáveis e passá-los também como forma de parâmetro num método. Sua sintaxe é mostrada na **Listagem 1**.

BOX 1. Generics

Generics é um recurso que permite ao desenvolvedor a criação de tipos parametrizados, ou seja, parâmetros passados para nossa classe ou método serão substituídos por um tipo. Esse tipo pode ser primitivo (Integer, Double, Boolean), estruturado (Records) ou complexo (TClass). É utilizado muito para criar listas de objetos, pois com seu uso ao passar um objeto que não pertence ao tipo especificado na lista, recebemos em erro em tempo de compilação e não em tempo de execução como ocorria nas versões do Delphi que não tínhamos suporte ao recurso.

BOX 2. Unicode

Unicode é um padrão que possibilita os dispositivos fazerem a representação e manipulação, de forma correta, em textos de qualquer sistema de escrita existente. Permite o uso, por exemplo, de caracteres japoneses, chineses, russos e etc.

BOX 3. Delegate

Delegates são referências diretas para um método através de uma variável ponteiro, ela possui o endereço de memória para aquele método. São semelhantes aos eventos do Delphi, onde temos referências a procedures.

BOX 4. Closure

Closures são blocos de códigos que podem ser executados a qualquer momento, guardando o contexto no qual eles foram criados, mesmo se a função que o criou saia do escopo.

Resumindo, para declarar um método anônimo precisamos:

- Informar se será um procedimento ou função
- Especificar o número de parâmetros
- Informar o tipo de retorno se for função

A **Listagem 2** a seguir, ilustra alguns exemplos.

Observando a própria RTL do Delphi, é possível notar alguns métodos anônimos. Por exemplo, a unit Sytem.SysUtils. Podemos fazer uso de qualquer destes métodos anônimos em nossas próprias aplicações, veja alguns exemplos na **Listagem 3**.

Ponteiros para Procedures e Functions

Para termos uma maior clareza sobre métodos anônimos, vamos revisar variáveis ponteiro para procedimentos, métodos de objetos e, além disso, agora temos também variáveis que fazem referência para métodos em nome. Veremos a seguir uma abordagem sobre as três formas.

O primeiro exemplo é o de variáveis que apontam para procedimentos e funções tradicionais do tipo procedural. O exemplo para este tipo é ilustrado na **Listagem 4.**

Observando o código da **Listagem 4,** primeiramente são definidos os tipos na seção Type, conforme as linhas 05 e 06. Após isso, implementamos um procedimento e uma função do mesmo tipo e parâmetros descritos anteriormente na sessão Type (linhas 07 a 14). Observe que a procedure MostraMensagem é autoexplicativa, apenas executando um comando Writeln para exibir uma mensagem, já na função DobraValor, é recebido um valor por parâmetro e multiplicado por 2, dobrando assim seu valor, conforme o próprio nome sugere. Na linha 15 temos a definição de duas variáveis apontando para as rotinas. O método A e B respectivamente (linhas 16 e 17). Na implementação atribuímos o procedimento e a função para estas variáveis, observe que estes devem ser do mesmo tipo, e conter o mesmo número de parâmetros. Para executarmos o código, basta chamar as variáveis pelo nome e, se necessário, passar os parâmetros, como mostrado na linha 22.

Listagem 1. Síntaxe de um método anônimo

```
01 [Nome] = reference to [procedure, function] [Param1: Tipo1, Param2:
   Tipo2, ParamN: TipoN]: [tipo retorno]
```

Listagem 2. Declaração de métodos anônimos

```
01 type
02  TExemploProcedure = reference to procedure;
03  TExemploFunction = reference to function: Integer;
04  TExemploProcedureParam = reference to procedure(x: Integer);
05  TExemploFunctionParam = reference to function(x: Integer): Integer;
```

Listagem 3. Exemplo de uso de métodos anônimos

```
01 TProc = reference to procedure;
02 TProc<T> = reference to procedure(Arg1: T);
03 TFunc<TResult> = referente to function: TResult;
04 TFunc<T, TResult> = referente to function(Arg1: T): TResult;
```

Listagem 4. Variáveis que apontam para procedimentos

```
01 program prjVarProcedure;
02 {$APPTYPE CONSOLE}
03 uses SysUtils;
04 type
05  TMostraMensagem = procedure;
06  TDobraValor = function (Value: Integer): Integer;
07 procedure MostraMensagem;
08 begin
09  Writeln('Executando procedimento');
10 end;

11 function DobraValor(Value: Integer): Integer;
12 begin
13  Result := Value * 2;
14 end;

15 var
16  A: TMostraMensagem;
17  B: TDobraValor;
18 begin
19  A := MostraMensagem;
20  A;
21  B := DobraValor;
22  Writeln(IntToStr(B(10)));
23  Readln;
24 end.
```

Ponteiros para Métodos

O segundo exemplo que veremos é o de variáveis que apontam para métodos (procedures e functions) de objetos de uma classe. Esta abordagem é muito utilizada na criação de eventos, mas por razões de simplicidade, iremos criar uma aplicação console tradicional. O código é mostrado na **Listagem 5**.

Listagem 5. Exemplo de ponteiro para métodos

```
01 program prjProcedureObject;
02 {$APPTYPE CONSOLE}

03 uses
04  System.SysUtils;

05 type
06  TProc = procedure of object;
07  TFunc = function (Value: Integer): string of object;

08  TClasseTeste = class
09  public
10   procedure MostraMensagem;
11   function GetDiaSemanaExtenso(Value: Integer): string;
12  end;

13 { TClasseTeste }

14 procedure TClasseTeste.MostraMensagem;
15 begin
16  Writeln('Mensagem');
17 end;

18 function TClasseTeste.GetDiaSemanaExtenso(Value: Integer): string;
19 begin
20  case Value of
21   1: Result := 'Domingo';
22   2: Result := 'Segunda';
23   3: Result := 'Terça';
24   4: Result := 'Quarta';
25   5: Result := 'Quinta';
26   6: Result := 'Sexta';
27   7: Result := 'Sábado';
28  else raise Exception.Create('Valor Inválido');
29  end;
30 end;

31 var
32  ClasseTeste: TClasseTeste;
33  Met1: TProc;
34  Met2: TFunc;

35 begin
36  ClasseTeste := TClasseTeste.Create;
37  try
38   Met1 := ClasseTeste.MostraMensagem;
39   Met2 := ClasseTeste.GetDiaSemanaExtenso;
40   Met1;
41   Writeln(Met2(3));
42  finally
43   FreeAndNil(ClasseTeste);
44  end;
55  Readln;
46 end.
```

Declaramos dois tipos de ponteiros para métodos de objetos (linhas 06 e 07). Um método aponta para um procedimento sem parâmetros e um aponta para uma função que recebe um parâmetro Integer e retorna uma String. A seguir, é criada classe de teste contendo dois métodos com a mesma declaração dos tipos, parâmetros e o mesmo retorno (linhas 08 a 12). Entre as linhas 14 a 30 podemos ver as respectivas implementações dos métodos. Note que o método MostraMensagem é exatamente como o da **Listagem 4**. Já o método GetDiaSemanaExtenso, simplesmente recebe os valores de 1 a 7 correspondentes aos dias da semana, retornando suas descrições por extenso.

Para realizar o teste, declaramos variáveis dos tipos criados e um objeto da classe teste (linhas 31 a 34). Instanciamos o objeto referente à classe (linha 36) e atribuímos aos ponteiros os métodos definidos anteriormente (linhas 38 e 39), em seguida, excutamos os métodos a partir dos ponteiros (linhas 40 e 41 respectivamente).

Métodos Anônimos

Métodos anônimos são formas de executar métodos através de chamadas sem nome. Esses métodos podem ser atribuídos às variáveis, ponteiros ou serem passados como parâmetro. É ideal para quem trabalha com muita iteração. Em outras palavras, com métodos anônimos criamos referências para "alguma coisa" sem saber o que é.

Para se trabalhar com métodos anônimos é necessário, antes de mais nada, criar um método executor. Este vai ter tudo aquilo que irá se repetir em determinada tarefa. Já aquela parte de código que "varia" é transformada num método anônimo.

Percorrendo um DataSet

Quem nunca precisou percorrer um DataSet em Delphi? Desabilitar os controles durante o percurso e restaurar a posição anterior no final? Pois é, os métodos anônimos podem nos ajudar nesta tarefa tão rotineira na vida de qualquer desenvolvedor.

Neste exemplo utilizamos o conjunto de componentes ClientDataSet, DataSource e DBGrid. Neste exemplo trabalharemos com dados em cache, sendo assim, o ClientDataSet está ligado ao arquivo "parts.xml" disponível na instalação do Delphi.

A **Listagem 6** exibe como ficaria o código da forma tradicional se desejássemos então fazer a soma da coluna ListPrice.

Listagem 6. Percorrendo DataSet da forma tradicional

```
01 procedure TfrmMain.btnSoma1Click(Sender: TObject);
02 var
03  Soma: Double;
04  Bookmark: TBookmark;
05 begin
06  Soma := 0.00;
07  CliDs.DisableControls;
08  try
09   Bookmark := CliDs.Bookmark;
10   CliDs.First;
11   while not CliDs.Eof do
12   begin
13    Soma := Soma + CliDs.FieldByName('ListPrice').AsCurrency;
14    CliDs.Next;
15   end;
16   CliDs.Bookmark := Bookmark;
17  finally
18   CliDs.EnableControls;
19  end;
20  txtSoma.Caption := FormatFloat('#,##0.00', Soma);
21 end;
```

São declaradas duas variáveis, uma do tipo TBookmark e outra como Double (linhas 03 e 04). A variável bookmark é um ponteiro para a posição do DataSet. Neste caso, a navegação visual do ClientDataSet é desabilitada (linha 07) e sua posição salva na variável anteriormente definida (linha 09). A seguir, na linha 10, o ClientDataSet é posicionado na primeira linha, realizando a soma da coluna ListPrice e adiantando suas posições até que chegue ao seu final (linhas 11 a 15). Na linha 16 o valor do Bookmark é atribuído novamente, posicionando-o no registro em que se encontrava, os controles visuais são reativados (linha 18) e por fim, o valor é ilustrado em um Label (linha 20).

No código apresentado é possível notar que toda vez que houver a necessidade de se percorrer um DataSet, a mesma operação irá ocorrer, deste modo, podemos aplicar dois conceitos interessantes. O primeiro é um dos princípios básicos da OO que diz que devemos encapsular o que varia. Neste caso, é abstraído o código que realiza a somatória dos valores. O segundo conceito é o reaproveitamento de código. E neste momento entram os métodos anônimos para nos ajudar na tarefa.

A primeira coisa a ser feita nesse caso é a declaração de um procedimento que receba como parâmetro um DataSet e uma Procedure, isso mesmo, um procedimento será passado como parâmetro! Veja a implementação sugerida na **Listagem 7**.

Listagem 7. Método que percorre o DataSet

```
01 procedure TfrmMain.PercorreDataSet(ADataSet: TDataSet; AProcedure: TProc);
02 var
03   Bookmark: TBookmark;
04 begin
05   ADataSet.DisableControls;
06   try
07     Bookmark := ADataSet.Bookmark;
08     ADataSet.First;
09     while not ADataSet.Eof do
10     begin
11       AProcedure;
12       ADataSet.Next;
13     end;
14     ADataSet.Bookmark := Bookmark;
15   finally
16     ADataSet.EnableControls;
17   end;
18 end;
```

Observe que ele é semelhante ao anterior, diferenciando-se apenas pelos parâmetros. A principal observação encontra-se na linha 11, onde temos a chamada do procedimento que foi passado como parâmetro. Na **Listagem 8** é demonstrada a parte mais interessante do exemplo.

Nas primeiras linhas temos a inicialização da variável Soma (linha 05). Em seguida, chamamos a procedure anteriormente criada "PercorreDataSet" (linha 06) e passamos o nosso DataSet como parâmetro. Neste momento, a soma é executada. No final temos a atribuição do resultado ao Label txtSoma (linha 11).

Podemos reaproveitar o mesmo código para popular um Memo com as descrições do DataSet de uma forma muito simples. Veja as adaptações realizadas na **Listagem 9**.

Listagem 8. Evento que soma fazendo uso do Método Anônimo

```
01 procedure TfrmMain.btnSoma2Click(Sender: TObject);
02 var
03   Soma: Double;
04 begin
05   Soma := 0.00;
06   PercorreDataSet(CliDs,
07     procedure
08     begin
09       Soma := Soma + CliDs.FieldByName('ListPrice').AsCurrency;
10     end);
11   txtSoma.Caption := FormatFloat('#,##0.00', Soma);
12 end;
```

Listagem 9. Evento que preenche o Memo fazendo uso do Método anônimo

```
01 procedure TfrmMain.btnPreencheMemoClick(Sender: TObject);
02 begin
03   PercorreDataSet(CliDs,
04     procedure
05     begin
06       memDescricoes.Lines.Add(CliDs.FieldByName('Description').AsString);
07     end);
08 end;
```

Implementamos dentro do procedimento (passado como parâmetro) o conteúdo da coluna "Description", adicionando-o ao Memo, conforme a linha 06. Agora não é mais necessário repetir o código quando precisarmos trabalhar com os campos do DataSet, bastando apenas chamar a procedure passando como parâmetro o código que muda. É claro que existem outras maneiras de implementar este recurso que não sejam por meio de métodos anônimos, porém, desta forma o código fica enxuto e de simples manutenção. A título de curiosidade, isto poderia ser feito utilizando a AOP (**BOX 5**).

BOX 5 . AOP

A Programação Orientada a Aspectos permite ao programador separar e organizar o código de acordo com a importância para a aplicação. Podemos separar e definir em aspectos o código que tem funções secundárias ao objeto, como gerenciamento de transações, logs e etc.

Percorrendo Strings

Outro exemplo onde podemos fazer uso de métodos anônimos é na comparação de dados de uma StringList. Imagine um cenário onde temos um Memo com um conjunto de dados e devemos exibir o maior valor, menor valor e a média de valores desta lista.

Programando da forma tradicional teríamos os eventos dos botões codificados conforme as **Listagens 10, 11 e 12**.

Os códigos são respectivamente são bem simples, não exigindo grandes explanações. Na **Listagem 10**, os itens da lista são contados um a um somando seus valores e atribuindo-os na variável soma. Ao final, é realizada a operação de Media, dividindo o resultado pelo número de iterações. Nas **Listagens 11** e **12** respectivamente, simplesmente a lista é percorrida, validando o maior e o menor valor dentro do conteúdo, exibindo-os ao final do processo.

Listagem 10. Realizando uma operação de media no StringList

```
01 procedure TfrmMain.btnMediaClick(Sender: TObject);
02 var i, Soma: Integer;
03   Media: Double;
04 begin
05   Soma := 0;
06   for i := 0 to memValores.Lines.Count -1 do
07   begin
08     Soma := Soma + StrToInt(memValores.Lines.Strings[i]);
09   end;
10   Media := Soma / memValores.Lines.Count;
11   txtResultado.Caption := 'Média: ' + FormatFloat('#,##0.00', Media);
12 end;
```

Listagem 11. Realizando uma operação de validação do maior numero em um StringList

```
01 procedure TfrmMain.btnMaiorClick(Sender: TObject);
02 var i, Maior: Integer;
03 begin
04   Maior := 0;
05   for i := 0 to memValores.Lines.Count -1 do
06   begin
07     if Maior < StrToInt(memValores.Lines.Strings[i]) then
08       Maior := StrToInt(memValores.Lines.Strings[i]);
09   end;
10   txtResultado.Caption := 'Maior: ' + IntToStr(Maior);
11 end;
```

Listagem 12. Realizando uma operação de validação do menor numero em um StringList

```
01 procedure TfrmMain.btnMenorClick(Sender: TObject);
02 var i, Menor: Integer;
03 begin
04   Menor := StrToInt(memValores.Lines.Strings[0]);
05   for i := 0 to memValores.Lines.Count -1 do
06   begin
07     if Menor > StrToInt(memValores.Lines.Strings[i]) then
08       Menor := StrToInt(memValores.Lines.Strings[i]);
09   end;
10   txtResultado.Caption := 'Menor: ' + IntToStr(Menor);
11 end;
```

Observando as listagens anteriores, novamente podemos notar a repetição de código, principalmente nos laços. Embora este seja um pequeno exemplo, em sistemas de grande porte esta situação é comum.

Com o objetivo de melhorar o código e separar o que se repete através de métodos anônimos, aplicaremos alguns dos conceitos vistos até aqui. Primeiro é implementado tudo o que se repete, que neste caso é o laço "for", através de um método chamado PercorreValores, conforme a **Listagem 13.**

A lista é percorrida e chamamos o procedimento que foi passando como parâmetro, passando o índice atual como parâmetro (linha 05). Note na chamada que recebemos um procedimento como parâmetro (ACalc) e o executamos a cada iteração do laço. Agora no lugar de chamarmos o laço, simplesmente executamos o método PercorreValores passando a lista de dados e um método sem nome com as instruções do que deve ser feito.

ProgressBar com métodos anônimos

Outra tarefa comum do dia a dia é a de configurarmos e processarmos uma ProgresBar. Tarefas repetitivas como configurar valor mínimo, máximo, passar para o próximo item, realizar o processamento. E que tal termos que fazer isso uma única vez e utilizarmos qualquer ProgressBar com qualquer tipo de processamento e qualquer mínimo e máximo? Conseguiremos isso com métodos anônimos. No exemplo a seguir vamos criar um método anônimo para realizar um processamento e atualizar automaticamente uma ProgressBar. O código encontra-se implementado nas **Listagens 14** e **15**.

Listagem 13. Método que recebe um procedimento anônimo como parâmetro

```
01 procedure TfrmMain.PercorreValores(AValores: TStrings; ACalc: TCalc);
02 var i: Integer;
03 begin
04   for i := 0 to memValores.Lines.Count - 1 do
05     ACalc(i);
06 end;
```

Listagem 14. Codificação do exemplo

```
01 type
02   TProcesso = reference to procedure(PassoAtual, Total: Integer);
03
04 procedure Processar(ProgressBar: TProgressBar; Processo: TProcesso);
05 var
06   i: Integer;
07 begin
08   if ProgressBar <> nil then
09   begin
10     ProgressBar.Step := 1;
11     for i := ProgressBar.Min to ProgressBar.Max do
12     begin
13       Processo(ProgressBar.Position, ProgressBar.Max);
14       ProgressBar.StepIt;
15     end;
16     ProgressBar.Position := 0;
17   end;
18 end;
```

Listagem 15. Utilização do exemplo

```
01 procedure TfrmMain.btnProcessarClick(Sender: TObject);
02 begin
03   Processar(Self.ProgressBar1,
04     procedure (PassoAtual, Total: Integer)
05     begin
06       StaticText1.Caption := Format('Passo %d de %d', [PassoAtual, Total]);
07       Sleep(100);
08     end);
09 end;
```

Foi declarada a assinatura do método anônimo na seção Type (linha 02), da **Listagem 14**, que receberá o passo atual e total que terá a barra de progresso. Também é criado o método Processar que recebe como parâmetro qualquer barra de progresso e um método anônimo, conforme a linha 04. Note que a linha 10 informa ao ProgressBar que seu incremento ocorrerá de 1 em 1, através da propriedade Step. A seguir, na linha 11, é iniciado um laço utilizando os valores mínimo e máximo da barra de progresso. Observe na linha 13 a execução do método anônimo. Ao final do processo o ProgressBar é zerado (linha 16).

Já no código da **Listagem 15** adicionamos um botão e em seu evento OnClick chamamos o método Processar alimentando os parâmetros e, em sua execução, simplesmente exibimos o passo atual num StaticText (linha 06). Vale salientar que aqui poderia ser feito qualquer outro tipo de processamento desejado em qualquer outra barra de progresso. O resultado é mostrado na **Figura 1**.

Com este exemplo, todas as vezes que necessitarmos realizar um processamento e exibir seu status atual, basta configurarmos o valor mínimo e máximo de nossa ProgressBar e chamar o método anônimo. Mais uma vez, este se mostra um poderoso recurso e que nos auxiliou a reduzir a quantidade e complexidade de código.

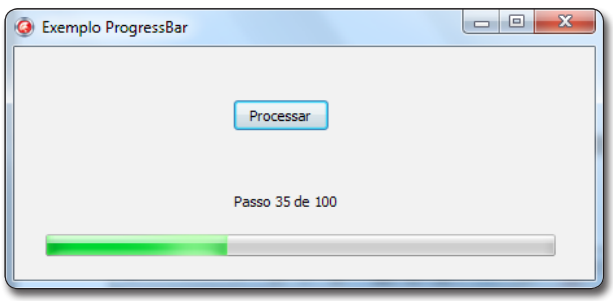

Figura 1. Aplicação em execução

Gerenciamento de Transações

Para o gerenciamento de transações em nível de banco de dados, temos uma sequência de passos bem conhecidos, que tratam-se da inicialização, execução de comandos de atualização no banco, e ações de acordo com o fluxo de execução. Neste caso, se tudo ocorre como o esperado, são confirmadas todas as operações através de um comando Commit, caso contrário é realizado o Rollback da aplicação, gerando-se em seguida a exceção propagada para quem chamou o método, conforme os códigos da **Listagem 16**.

Mais uma vez percebemos a duplicidade. Uma forma de contornarmos isso é criando um método anônimo. Em ambos os casos temos a abertura da transação, confirmação, cancelamento e propagação da exceção se ocorrer um erro. Novamente a exemplo de apresentar os conceitos já vistos, vamos extrair a parte em comum e criar um método anônimo com ele. Podemos usar o tipo TProc já declarado na System.SysUtils para o exemplo. Veja o novo código aplicando métodos anônimos na **Listagem 17.**

Foi definido o método ProcessaEmTransacao (linha 01). Na linha 03 a transação é iniciada, tendo a execução do método anônimo (recebido por parâmetro) realizada logo em seguida, na linha 05. Caso o código ocorra sem problemas, automaticamente é realizado o Commit na linha 06, caso contrário, é realizado um Rollback revertendo as alterações e expondo a exceção (linhas 08 e 09). Com este método anônimo é possível fazer qualquer operação dentro de uma transação, bastando chamar a procedure ProcessaEmTransacao passando o processamento desejado como parâmetro. Observe que neste caso é realizada a simulação de uma operação de Insert (linhas 12 a 20) e Update (linhas 21 a 28). Atente para estética do código, não tanto para a quantidade de linhas codificadas. Ele ficou bem mais elegante e claro.

Listagem 16. Repetições de código no gerenciamento de transações

```
01 procedure Insert;
02 begin
03  BeginTransaction;
04  try
05   Operacao1;
06   Commit;
07  except
08   Rollback;
09   raise;
10  end;
11 end;

13 procedure Update;
14 begin
15  BeginTransaction;
16  try
17   Operacao2;
18   Commit;
19  except
20   Rollback;
21   raise;
22  end;
23 end;
```

Listagem 17. Gerenciamento de transações com métodos anônimos

```
01 procedure ProcessaEmTransacao(Proc: TProc);
02 begin
03  BeginTransaction;
04  try
05   Proc;
06   Commit;
07  except
08   Rollback;
09   raise;
10  end;
11 end;

12 procedure Insert;
13 begin
14  ProcessaEmTransacao(
15   procedure
16   begin
17    Operacao1;
18   end);
20 end;

21 procedure Update;
22 begin
23  ProcessaEmTransacao(
24   procedure
25   begin
26    Operacao2;
27   end);
28 end;
```

A adoção desta sintaxe ao invés da outra também nos dá maior segurança. Imagine se esquecemos de abrir a transação, confirmar ou cancelar em caso de erro. Teríamos erros em tempo de execução, ou pior, perda e inconsistência de dados. Outro problema seria esquecer de propagar a exceção, o que passaria a impressão que deu tudo certo no processamento, quando na verdade, nada teria sido processado.

Agora imagine se no futuro resolvermos fazer um log de transações, gravando as operações no banco de dados. A única

parte que necessitaria de alterações seria o método anônimo e, automaticamente, todo o resto do sistema estaria consumindo o novo recurso. As vantagens são inúmeras.

Threads

Todas as aplicações Delphi executam pelo menos dentro de uma thread, ou seja, a thread principal. Em aplicações que executam tudo dentro de única thread, tem seu fluxo realizado de forma sequencial.

Quando temos processos demorados, a aplicação fica congelada, pois a thread está em uso e a VCL não pode tratar os eventos relacionados aos cliques de mouse ou comandos do teclado. Para visualizarmos esta situação, basta codificar, por exemplo, um comando Sleep(4000) no evento OnClick de um Button. Ao testar a aplicação, vemos que ela fica congelada no clique do botão, se tentarmos arrastar a janela, por exemplo, a ação pode demorar um pouco para ocorrer, pois a thread principal está aguardando aquele tempo que especificamos na instrução Sleep.

Para resolver este tipo de problema, usa-se a criação de threads, e a forma mais tradicional de se trabalhar com elas é criar uma classe que herde da classe TThread e sobrescrever o seu método Execute, fazendo com que o processamento seja separado da thread principal, conforme mostrado na **Listagem 18**.

Listagem 18. Exemplo de Thread

```
01 unit uExemploThread;

02 interface

03 uses System.Classes;

04 type
05  TExemploThread = class(TThread)
06  protected
07    procedure Execute; override;
08  end;

09 implementation

10 procedure TExemploThread.Execute;
11 begin
12  inherited;
13  Sleep(400);
14 end;

15 end.

16 procedure TForm1.btnThreadNormalClick(Sender: TObject);
17 var
18  Thread: TExemploThread;
19 begin
20  Thread := TExemploThread.Create(True);
21  try
22    Thread.Start;
23  finally
24    Thread.Free;
25  end;
26 end;
```

Observe que foi realizado o processo anteriormente mencionado. Criamos assim a classe TExemploThread descendente de TThread (linhas 05 a 08), sobrescrevendo o método Execute, através da cláusula override (linha 07). Neste momento, é implementado o código referente ao Sleep que fazia com que a VCL congelasse (linhas 10 a 14). Por último, declaramos em a variável Thread (TExempThread) em uma implementação de um Button (linha 18), a seguir criamos o objeto (linha 20) e executamos seu código, através do comando Start (linha 22). Isto faz com que automaticamente o código dentro do método Execute seja acionado. Por fim, a thread é liberada da memória (linha 24).

Vale salientar que podemos fazer a mesma tarefa usando um método anônimo definido na classe da própria TThread, o Create-AnonymousThread. Ele recebe como parâmetro uma procedure com o processamento que desejamos fazer, o restante é tudo feito internamente. Veja o exemplo na **Listagem 19**.

Listagem 19. Uso da Thread através de métodos anônimos

```
01 procedure TForm1.btnThreadMetAnonimoClick(Sender: TObject);
02 begin
03  TThread.CreateAnonymousThread(
04    procedure
05    begin
06      Sleep(4000);
07    end);
08 end;
```

Simplesmente é criada a procedure que realiza o procedimento de Sleep, sendo automaticamente passado como parâmetro para a Thread (linhas 03 a 07).

Multicast Events

Antes de entrarmos no assunto de Multicast Events, vamos falar dos Eventos tradicionais do Delphi. Quando damos um duplo clique num botão, é criado um manipulador para o evento OnClick, que cria automaticamente uma procedure. O evento OnClick é um ponteiro para uma implementação. Esse ponteiro possui um tipo, que define a assinatura do método e os parâmetros que ele deve conter.

Multicast Events é a forma de referenciarmos vários métodos num único evento. No Delphi quando temos um controle na tela e desejamos manipular o evento OnClick, só conseguimos vincular uma única execução do procedimento a este evento. O Multicast Event é um recurso existente na plataforma .NET e podemos através de métodos anônimos, simulá-los no Delphi.

Para a realização do Multicast, o evento deve receber não um, mas uma lista de procedures que desejamos executar já em tempo de projeto. Ao chamar o evento, deve-se chamar cada um dos métodos, um a um, lembrando que os eventos sempre recebem uma lista de procedures que contém a mesma quantidade de parâmetros.

Conclusão

Aplicando métodos anônimos corretamente em nossas aplicações, temos um código mais limpo e de fácil manutenção. Torna-se prazeroso trabalhar quando vemos que nossa aplicação está clara, objetiva e utiliza bem todos os recursos da linguagem. Um bom programador deve conhecer bem todos os recursos que a linguagem lhe oferece, e saber aplicá-los da melhor maneira.

Notamos que os recursos podem ser utilizados sempre que há códigos redundantes. Como quando repetimos muitas vezes um mesmo laço por exemplo.

Que este artigo tenha contribuído para que os sistemas escritos ou migrados a partir da versão 2009 façam uso deste fantástico recurso oferecido pela nossa linguagem.

Autor

Filipe Dalepiane
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/

Analista Desenvolvedor Delphi a 5 anos, utilizando arquitetura multicamadas, com os Banco de Dados Firebird e DB2. Colunista da revista Clube Delphi. Adepto de boas práticas de programação, modelagem UML, Orientação a Objetos e Design Patterns. Graduando Ciência da Computação pela UNIJUÍ.

Links:

Generics e Métodos Anônimos
http://cc.embarcadero.com/item/26347

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Clube Delphi

CONHEÇA ALGUNS DOS CURSOS:

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :
www.devmedia.com.br/cursos/delphi
(21) 3382-5038

DevMedia

Padrões Estruturais em Delphi

Aplicando Padrões Adapter, Bridge, Composite, Decorator e Façade

Resumo DevMan

Padrões Estruturais em Delphi:

Padrões de projeto são soluções prontas e reutilizáveis para problemas recorrentes da orientação a objetos, entre eles os principais são: forte acoplamento entre classes, dificuldade de reutilização de componentes, falta de modularização, duplicação de código e altos custos com manutenção. Os padrões de projeto são apresentados no livro "Design Patterns: Elements of Reusable Object-Oriented Software", dos autores E. Gamma, R. Helm, R. Johnson e J. Vlissides, mais conhecidos como Padrões GoF (Gang of Four), ver referências bibliográficas na sessão **Links**. No artigo da edição 152 da Revista ClubeDelphi foram abordados os padrões da categoria criacionais, incluindo Abstract Factory, Factory Method, Singleton e Builder, que tinham por finalidade resolver problemas com forte acoplamento de classes devido a criação de objetos concretos. Neste artigo, vamos abordar alguns padrões do grupo Estrutural, que têm por finalidade mostrar como formar objetos a partir de funcionalidades de outros objetos, seja por herança de abstrações, associações e composição. Os padrões apresentados neste artigo são o Adapter, Bridge, Composite, Decorator e Façade.

Em que situação o tema é útil:

Padrões de projeto estruturais são úteis para criar softwares mais fáceis de serem mantidos, evoluídos, promovendo reutilização de código, uso de boas práticas de programação orientada a objetos, redução de custos com manutenção, já que permitem que alterações em um determinado sistema não quebrem outras funcionalidades, reduzindo a dependência entre classes, através do uso exaustivo de abstrações. Dos padrões tratados neste artigo, o Adapter serve para permitir a comunicação entre duas classes com interfaces distintas; Bridge serve para separar totalmente a interface de um objeto de sua implementação concreta; Composite permite criar composições de objetos; Decorator permite adicionar funcionalidades a um objeto sem usar herança estática, usando herança de "caixa-preta", através da implementação por delegação; Façade ajuda a tornar um subsistema de classes mais fácil de utilizar. Como na edição anterior, primeiramente será apresentada uma descrição formal de cada padrão segundo GoF, depois um exemplo prático em Delphi.

A intenção do padrão Adapter, também conhecido como Wrapper, é converter a interface de uma classe em outra interface esperada pelos clientes. Permite que classes trabalhem em conjunto, pois de outra forma não poderiam, devido ao fato de terem interfaces incompatíveis. O padrão pode ser usado quando se quiser usar uma classe existente, mas sua interface não corresponder à interface que necessita; ou quando quiser criar uma classe reutilizável que coopere com classes não relacionadas ou não previstas, ou seja, classes que não necessariamente tenham interfaces compatíveis; você precisa usar várias subclasses existentes, porém, quando for impraticável adaptar essas interfaces criando subclasses para cada uma. O padrão traz alguns benefícios. Um adaptador de classe adapta Adaptee a Target através do uso efetivo de uma classe Adapter concreta, com isso, permite a Adapter substituir algum comportamento do Adaptee, uma vez que Adapter é uma subclasse de Adaptee. Um adaptador de objeto permite que um único Adapter trabalhe com muitos Adaptees. O Adapter também pode acrescentar funcionalidades a todos os Adaptees de uma só vez.

Os participantes do padrão de projeto são:

- Target – define a interface específica do domínio que o Client usa;
- Client – colabora com objetos compatíveis com a interface de Target;

• Adaptee – define uma interface existente que necessita ser adaptada;
• Adapter – adapta a interface do Adaptee à interface do Target;

Para entender melhor o padrão Adapter, vamos recorrer a alguns objetos do mundo real. Por exemplo, digamos que quiséssemos conectar nosso notebook em uma tomada para recarregar sua bateria. Ao entrar em uma sala e encontrar um ponto livre, reparamos que a tomada fêmea segue o novo padrão brasileiro NBR 14136, enquanto nosso notebook tem um cabo com plug macho no padrão norte americano NEMA 5. Ou seja, estamos com problemas para conectar dois objetos, pois eles possuem **interfaces incompatíveis.** A solução para este problema no mundo real é fácil de ser resolvida, precisamos de um **Adaptador**, ou Adapter. Ele vai se encarregar de converter o a interface do Adaptee (o plug do nosso notebook) em um formato que o Target entenda (o alvo, a tomada, onde realmente queremos conectar). A **Figura 1** mostra um exemplo de como seriam nossos personagens do padrão de projeto em objetos do mundo real.

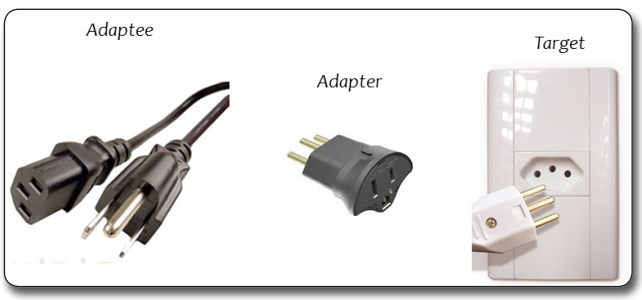

Figura 1. Adaptador no mundo real conecta duas interfaces distintas

No mundo orientado a objetos, um Adapter é frequentemente encontrado em muitas situações. No Delphi, por exemplo, temos dezenas de aplicações deste padrão. Vamos observar a arquitetura do novo DataSnap. No modelo antigo do DataSnap (e primeiras versões do MIDAS), tínhamos um componente de conexão remoto para cada tipo de protocolo que nosso servidor de aplicação pudesse trabalhar. Por exemplo, um DCOMConnection era utilizado no cliente para conectar a um servidor DCOM, MTS ou COM+. A partir do Delphi 6, com o suporte a Web Services, foi possível utilizar um SOAPConnection para conectar em um servidor de aplicação baseado em SOAP (Simple Object Access Protocol). E poderíamos citar outros tipos de conexão, como CorbaConnection, WebConnection, SocketConnection e etc. Um ClientDataSet pode então se conectar a qualquer um desses componentes concretos para obter dados de diferentes tipos de servidores, independente do protocolo que usem. Para isso, basta aponta-los para a propriedade RemoteServer de ClientDataSet (**Figura 2**), que é do tipo TCustomRemoteServer, classe base para todos os componentes de conexão (SOAPConnection, DCOMConnection, SocketConnection, WebConnection, CorbaConnection etc.). Com o novo DataSnap, do Delphi 2009 em diante, uma grande mudança foi feita, ao invés de termos um componente com uma implementação concreta para cada protocolo de comunicação DataSnap, passamos a usar um único componente de conexão, o próprio SQLConnection do DBExpress. O código de acesso dependente de protocolo passou a residir então em um driver do DBExpress (DLL). Agora vem o problema, se tentarmos conectar um ClientDataSet em um SQLConnection pela propriedade RemoteServer, o que acontece? O SQLConnection nem aparece na lista, pois este descende de TCustomConnection, e não de TCustomRemoteServer. Como é impossível usar herança múltipla no Delphi, pelo menos de forma estática, não há uma forma de resolver o problema sem causar uma mudança profunda na VCL, modificar a classe base de um dos tipos citados, o que iria ferir princípios fundamentais da orientação a objetos (e uma reação em cadeia de alterações em todo framework). Então, um padrão de projeto foi aplicado para resolver o problema de forma elegante. A classe TDSProviderConnection funciona como uma "ponte" entre um ClientDataSet e um SQLConnection, fazendo com que o SQLConnection "finja" ser um RemoteServer para o ClientDataSet (**Figura 3**), ou seja, é um autêntico Adapter, como sua própria declaração descrita na **Listagem 1** o identifica.

Listagem 1. Declaração da classe TDSProviderConnection

```
TDSProviderConnection = class(TCustomRemoteServer, IAppServer...
```

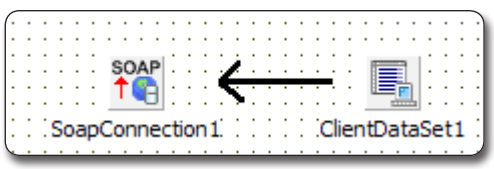

Figura 2. ClientDataSet se conecta a um SoapConnection via RemoteServer

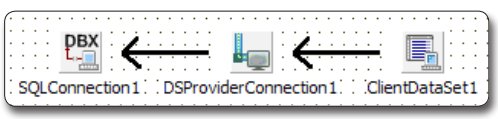

Figura 3. DSProviderConnection é um Adapter para as classes TCustomRemoteServer (Target) e TCustomConnection (Adaptee)

Na **Listagem 2** vemos um exemplo de uso do Adapter em Delphi, cujas classes podem ser vistas no diagrama de classes UML da **Figura 4**. Aqui é definida uma classe para cada um dos participantes do padrão de projeto, o Adaptee (linhas 04 a 09), Target (linhas 10 a 14) e Adapter (linhas 17 a 26). Um método "Conectar" simula a chamada de métodos de interfaces incompatíveis. Note na linha 17 que Adapter herda de Target (nosso alvo), ou seja, possui uma interface compatível. Por delegação, ele repassa a chamada aos métodos de um Adaptee incompatível, que é nosso objeto adaptado (linha 39 a 43).

A **Listagem 3** mostra um exemplo de uso do framework. Repare que declaramos um tipo da classe Target (Alvo) na linha 08, porém, na hora de "conectarmos", instanciamos uma classe compatível com Target, ou seja, com a interface compatível, obtida através de herança (conforme as linhas 12 e 13).

Listagem 2. Adapter

```
01 unit uFramework;

02 interface

03 type
04  // Adaptee
05  Adaptado = class
06   // interface diferente de request
07  public
08   procedure Conectar();
09  end;

10  // Target
11  Alvo = class
12  public
13   procedure Conectar(); virtual;
14  end;

15  // transforma Adaptee num Target...
16  // Adapter
17  Adaptador = class(Alvo)
18  // composição simples
19  private
10   _adaptee: Adaptado;
21   // delegação simples, repassa chamada ao adaptee
22  public
23   procedure Conectar(); override;
24   constructor Create();
25   destructor Destroy(); override;
26  end;

27 implementation

28 { Adaptee }

29 procedure Adaptado.Conectar();
30 begin
31  WriteLn('Método Conectar de Adaptee(Adaptado) chamado');
32 end;

33 { Target }

34 procedure Alvo.Conectar();
35 begin
36  WriteLn('Método Conectar de Target(Alvo) chamado');
37 end;

38 { Adapter }

39 procedure Adaptador.Conectar();
40 begin
41  // chama por delegação
42  _adaptee.Conectar();
43 end;

44 constructor Adaptador.Create();
45 begin
46  // cria objeto a ser adaptado
47  _adaptee := Adaptado.Create();
48 end;

49 destructor Adaptador.Destroy();
50 begin
51  // destroi adaptado
52  _adaptee.Free();
53  inherited;
54 end;

55 end.
```

Listagem 3. Usando um Adapter

```
01 program AdapterApp;

02 {$APPTYPE CONSOLE}

03 {$R *.res}

04 uses
05  System.SysUtils,
06  uFramework in 'uFramework.pas';

07 var
08  _Alvo: Alvo;
09 begin

10  // não pode-se usar "Adaptado.Create()"
11  // devido a interface incompatível
12  _Alvo := Adaptador.Create();
13  _Alvo.Conectar();
14  Readln;
15 end.
```

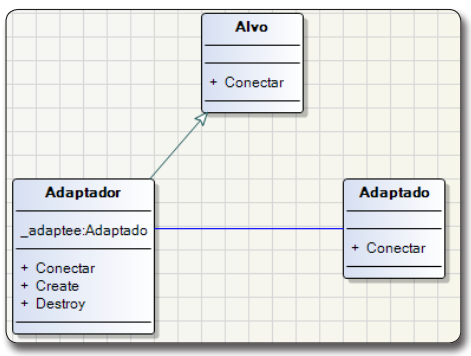

Figura 4. Estrutura de classes do exemplo usando Adapter

Bridge

A intenção do padrão Bridge é desacoplar uma abstração de sua implementação, de modo que os dois possam variar independentemente. O padrão pode ser usado para evitar o vínculo permanente entre uma abstração e sua implementação, por exemplo, quando a implementação deve ser alterada em tempo de execução. Tanto as abstrações como as suas implementações tiverem de ser

extensíveis por meio de subclasses, nesse caso, permite combinar diferentes abstrações e implementações e estendê-las independentemente. Mudanças na implementação de uma abstração não tem impacto sobre clientes. Ou ainda, para compartilhar uma mesma implementação entre vários objetos. Entre os benefícios trazidos pelo padrão, podemos citar o desacoplamento da interface da implementação, melhoria na extensibilidade e ocultação de detalhes de implementação dos clientes.

Os participantes do padrão de projeto são:

- Abstraction – define a interface da abstração; mantém uma referência para o objeto do tipo Implementor;
- Refined Abstraction – estende a interface definida por Abstraction;
- Implementor – define a interface para as classes de implementação;
- Concrete Implementor – implementa a interface de Implementor e define sua implementação concreta;

O Pascal, diferentemente do C#, como todos sabem, sempre teve uma separação entre interface e implementação em nível de unit, mesmo que não use classes. O que é declarado em uma interface, deve obrigatoriamente ser implementado em uma sessão implementation. Se a interface varia, a implementação também muda. Se uma nova implementação for necessária, talvez seja necessário modificar alguma coisa na interface, o que desencadeia uma recompilação de dependências. O mesmo vale para classes que implementam abstrações e interfaces. Uma interface deveria ser imutável, segundo princípios básicos da orientação a objetos. Se uma interface muda, como por exemplo, remoção / adição de um método ou modificação de uma assinatura, todas as classes concretas que a implementam precisam ser modificadas. Quando dizemos que uma classe herda de uma classe base e implementa um determinado conjunto de interfaces, essa é uma ligação estática, uma herança de classe e interface quase que imutável, o que chamamos de herança de "caixa-branca". A herança, apesar de parecer ser um dos melhores recursos da orientação a objetos, expõe muito publicamente o que herda, e isso é imutável. Ao contrário, uma implementação de herança por "caixa-preta" e delegação, permite esconder a classe "base", simulando herança múltipla e encapsulando quem são as classes base, que podem ser trocadas a qualquer momento sem conhecimento público. O padrão Bridge faz exatamente isso, de forma bem mais radical: além de não fazer herança de caixa branca entre interface e implementação, é possível "injetar" de fora uma implementação compatível com o que a classe precisa, que fará uso dos recursos "herdados" por delegação. Ou seja, é uma típica implementação por delegação, sem herança entre abstração e implementação.

Existem várias classes da VCL / FMX do Delphi que delegam a sua funcionalidade para algo "herdado" por delegação. Por exemplo, um TSimpleDataSet do DBExpress permite que o desenvolvedor utilize um único componente para conectar em um banco de dados, executar comandos SQL, fazer cache e resolver atualizações, de forma a eliminar a necessidade de se usar quatro componentes distintos para essa tarefa, os já conhecidos SQLConnection, SQLDataSet, ClientDataSet e DataSetProvider. Não, o TSimpleDataSet não herda destas quatro classes, porque, mais uma vez, Delphi não suporta herança múltipla, mas o componente consegue realizar o papel de todos eles. Ao invés disso, usa a herança de caixa-preta, os componentes são instanciados internamente, e quando fazemos uma chamada a determinado método, o mesmo é resolvido internamente por delegação. Agora repare algo interessante: se for necessário usar um SQLConnection externo ao componente, que não o criado internamente, pode-se apontar a sua propriedade Connection. Em outras palavras, está "injetando" externamente uma implementação em uma interface já definida pela propriedade Connection. Se existir outra implementação de SQLConnection, compatível com a interface, essa também pode ser atribuída a ela. Isso funciona de forma muito semelhante a um padrão de projeto Bridge.

Nas **Listagens 4** e **5** temos um exemplo de um framework que utiliza o padrão Bridge. Seu diagrama de classes encontra-se na **Figura 5**.

Note claramente que temos duas classes distintas, uma para definir uma abstração e outra para definir uma implementação (linhas 04 a 17). Agora repare no detalhe mais importante: a implementação não herda da abstração. Ou seja, se a interface de uma abstração for modificada, sua implementação não precisa obedecer explicitamente esta mudança, e vice-versa. A implementação é "injetada de fora", através do método setImplementor (linha 12). Basicamente, a classe recebe uma implementação concreta do que deve fazer, e guarda a referência a esse objeto em uma variável privada interna (_implementor – linha 13). Dessa forma, quando necessário, a classe aciona algum método de _implementor, que por delegação, efetua as operações desejadas.

A **Listagem 5** demonstra a implementação dos objetos concretos acionados por delegação.

A **Listagem 6** mostra como esse framework pode ser utilizado. Basicamente criamos uma "forma concreta de uma abstração" (linha 11), que não tem implementação, e depois dizemos quais são as classes que contêm o código funcional (linhas 12 a 23). Mais uma vez, repare que não há herança entre abstração e implementação.

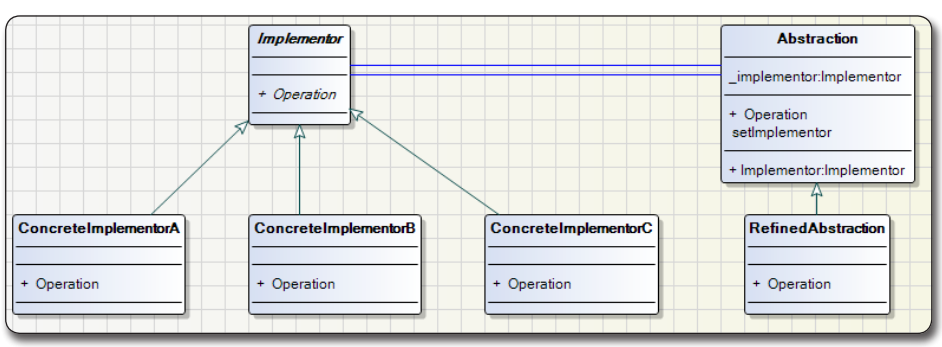

Figura 5. Estrutura de classes do exemplo usando Bridge

Composite

Este padrão tem por intenção compor objetos em estruturas de árvore para representar hierarquias em parte ou como um todo. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme. Utilizado sempre que é necessário representar elementos que são compostos por outros elementos similares. Por exemplo, em interfaces gráficas um elemento gráfico pode ser constituído pela composição de vários outros elementos gráficos. Uma janela pode conter um ou mais ícones, uma caixa de texto e vários outros elementos gráficos - até mesmo outra janela. Considerando que uma determinada hierarquia de classes indicasse Control como a superclasse comum a

Listagem 4. Bridge – declaração

```
01 unit uFramework;

02 interface

03 type

04  // "Implementor"
05  Implementor = class abstract
06  public
07    procedure Operation(); virtual; abstract;
08  end;

09  // "Abstraction"
10  Abstraction = class
11  private
12    procedure setImplementor(const Value: Implementor);
13    protected _implementor: Implementor;
14  public
15    property Implementor: Implementor write setImplementor;
16    procedure Operation(); virtual;
17  end;

18  // "RefinedAbstraction"
19  RefinedAbstraction = class(Abstraction)
20  public
21    procedure Operation(); override;
22  end;

23 { Abstraction }

24 procedure Abstraction.Operation();
25 begin
26  // implementação por delegação
27  _implementor.Operation();
28 end;

29 procedure Abstraction.setImplementor(const Value: Implementor);
30 begin
31  self._implementor := Value;
32 end;

33 { RefinedAbstraction }

34 procedure RefinedAbstraction.Operation();
35 begin
36  // implementação por delegação
37  self._implementor.Operation();
38 end;
39 ...
```

Listagem 5. Continuação do padrão Bridge - implementação

```
01 ...
02  // "ConcreteImplementorA"
03  ConcreteImplementorA = class(Implementor)
04  public
05    procedure Operation(); override;
06  end;

07  // "ConcreteImplementorB"
08  ConcreteImplementorB = class(Implementor)
09  public
10    procedure Operation(); override;
11  end;

12  // "ConcreteImplementorC"
13  ConcreteImplementorC = class(Implementor)
14  public
15    procedure Operation(); override;
16  end;

17  // "ConcreteImplementorD"
18  ConcreteImplementorD = class(Implementor)
19  public
20    procedure Operation(); override;
21  end;

22 implementation

23 { ConcreteImplementorB }

24 procedure ConcreteImplementorB.Operation();
25 begin
26  Writeln('Operação de Implementador Concreto B chamado');
27 end;

28 { ConcreteImplementorA }

29 procedure ConcreteImplementorA.Operation();
30 begin
31  Writeln('Operação de Implementador Concreto A chamado');
32 end;

33 { ConcreteImplementorC }

34 procedure ConcreteImplementorC.Operation();
35 begin
36  Writeln('Operação de Implementador Concreto C chamado');
37 end;

38 { ConcreteImplementorD }

39 procedure ConcreteImplementorD.Operation();
40 begin
41  Writeln('Operação de Implementador Concreto D chamado');
42 end;

43 end.
```

todas classes que representassem os elementos gráficos atômicos, a classe Form seria representada como uma classe que contém zero (0) ou mais elementos gráficos. É importante observar que será responsabilidade do objeto composto, para cada método a ser aplicável à lista de objetos que possui, implementá-lo de maneira repetitiva. Por exemplo, no caso da hierarquia de elementos gráficos exemplificada, suponha que um objeto cliente ative o método Draw de um objeto do tipo Form. Este método deverá ser capaz de ativar os métodos Draw de cada um dos objetos que ele contém. Desta maneira, será possível interagir com uma composição de objetos da mesma forma que se interage com objetos individuais. Entre os benefícios trazidos pelo uso do padrão, podemos citar a definição de hierarquia de classes que consistem de objetos primitivos e objetos compostos, podendo assim compor objetos mais complexos. Sempre que o código do cliente esperar um objeto primitivo, poderá também aceitar um composto. Além disso, o padrão torna o cliente simples, pois pode tratar estruturas diferentes de maneira uniforme (não importa se é um tipo folha ou composto). Torna também mais fácil a adição de novas espécies de componentes.

Os participantes do padrão de projeto são:

• Component – declara a interface para os objetos na composição; implementa comportamento padrão para a interface comum; declara interface para gerenciar filhos;

Listagem 6. Usando um Bridge

```
01 program Bridge;

02 {$APPTYPE CONSOLE}

03 {$R *.res}

04 uses
05   System.SysUtils,
06   uFramework in 'uFramework.pas';

07 var
08   // Programação para uma interface / abstração
09   ab: Abstraction;
10 begin
11   ab := RefinedAbstraction.Create();
12   // Configura implementador e faz chamada de operação
13   ab.Implementor := ConcreteImplementorA.Create();
14   ab.Operation();
15   // Troca implementador e faz chamada de operação
16   ab.Implementor := ConcreteImplementorB.Create();
17   ab.Operation();
18   // Troca implementador e faz chamada de operação
19   ab.Implementor := ConcreteImplementorC.Create();
20   ab.Operation();
21   // Troca implementador e faz chamada de operação
22   ab.Implementor := ConcreteImplementorD.Create();
23   ab.Operation();
24   Readln;
25 end.
```


CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :
www.devmedia.com.br/curso/javamagazine
(21) 3382-5038

• Leaf – representa objetos "folha" na composição;
• Composite – define comportamentos para os componentes que têm filhos; implementa operações relacionadas com os filhos;
• Client – manipula objetos na composição através da interface de Component;

O Delphi implementa o padrão Composite na VCL desde a sua versão 1. Vamos tomar por base a hierarquia de classes da VCL. TComponent é a classe base para todos os componentes visuais e não visuais. TControl, descendente direto de TComponent, serve de base para controles do Windows (TWinControl), como botões, edits e formulários. TComponent implementa um mecanismo para gerenciar componentes filhos, de modo a utilizar o conceito de Owner para alocar e liberar outros controles, com o conceito de componente proprietário. Ele guarda internamente uma referência a todos os outros componentes que ele é pai, no atributo FComponents: TList<TComponent>. Através de métodos públicos como "InsertComponent(AComponent: TComponent)" e "RemoveComponent(AComponent: TComponent)" é possível adicionar ou remover componentes dessa lista. Um mecanismo de notificação, que usa o padrão de projeto Observer (um padrão comportamental), se encarrega de avisar componentes associados quando algo acontece com o componente, como por exemplo, ser removido do formulário.

Todas as classes visuais descendem então em algum momento de TControl, que descende de TComponent. Ou seja, TEdit, TButton, TDBGrid são TComponents, como todos sabem. Agora repare um formulário (TForm), ele também em algum momento herda de TControl, porque ele é visual, e em mais baixo nível, de TComponent. Um formulário pode visualmente conter outros TControls, como Edits, Buttons e Grids. Ele é tipicamente uma classe simples (um Form em branco não faz muita coisa) que, para prover todas suas funcionalidades (um cadastro, por exemplo), deve ser formado por outros controles, formando o que conhecemos na orientação a objetos de composição. Então, um TForm é um TComponent, como também contém outros TComponents. Essa é a implementação real de um típico padrão Composite.

As **Listagens 7** e **8** mostram um framework de classes, visto no diagrama da **Figura 6**.

Na **Listagem 7** definimos os participantes do padrão Composite, basicamente foi simulado o framework de classes da VCL, como o componente base (linhas 04 a 14), formulários (linhas 15 a 23), botões, labels e etc. Um formulário funciona como a composição, ou seja, pode conter outros controles, chamados aqui de "nós folhas" (Leafs – linhas 24 a 58). Neste exemplo, os nós Leafs não podem funcionar como uma composição, apenas o formulário.

Listagem 7. Composite - declaração

```
01 unit uFramework;

02 interface

03 uses Generics.Collections;

04 type
05  // Component
06  Component =  class abstract
07  protected
08    _name: string;
09  public
10    procedure Add(c: Component); virtual; abstract;
11    procedure Remove(c: Component);virtual; abstract;
12    procedure Display(); virtual; abstract;
13    constructor Create(name: string); virtual;
14  end;

15  // Composição
16  Formulario = class (Component)
17  private
18    _childrens : TList<Component>;
19  public
20    procedure Add(c: Component); override;
21    procedure Remove(c: Component); override;
22    procedure Display(); override;
23  end;

24  // Leaf (folha)
25  TextBox = class(Component)
26  public
27    procedure Add(c: Component); override;
28    procedure Remove(c: Component); override;
29    procedure Display(); override;
30  end;

31  // Leaf (folha)
32  Button = class(Component)
33  public
34    procedure Add(c: Component); override;
35    procedure Remove(c: Component); override;
36    procedure Display(); override;
37  end;

38  // Leaf (folha)
39  &Label = class(Component)
40  public
41    procedure Add(c: Component); override;
42    procedure Remove(c: Component); override;
43    procedure Display(); override;
44  end;

45  // Leaf (folha)
46  Panel = class(Component)
47  public
48    procedure Add(c: Component); override;
49    procedure Remove(c: Component); override;
50    procedure Display(); override;
51  end;

52  // Leaf (folha)
53  Image = class(Component)
54  public
55    procedure Add(c: Component); override;
56    procedure Remove(c: Component); override;
57    procedure Display(); override;
58  end;
```

Listagem 8. Composite - implementações

```
01 ...
02 implementation

03 { Component }

04 constructor Component.Create(name: string);
05 begin
06   self._name := name;
07 end;

08 { Formulario }

09 procedure Formulario.Add(c: Component);
10 begin
11   if _childrens = nil then
12     _childrens := TList<Component>.Create();
13   _childrens.Add(c);
14 end;

15 procedure Formulario.Display();
16 var
17   c: Component;
18 begin
19   for c in _childrens do
20     c.Display();
21 end;

22 procedure Formulario.Remove(c: Component);
23 begin
24   _childrens.Remove(c);
25 end;

26 { TextBox }

27 procedure TextBox.Add(c: Component);
28 begin
29   // não adiciona porque não é composição
30 end;

31 procedure TextBox.Display();
32 begin
33   WriteLn(_name);
34 end;

35 procedure TextBox.Remove(c: Component);
36 begin
37   // não remove porque não é composição
38 end;

39 { Button }

40 procedure Button.Add(c: Component);
41 begin
42   // não adiciona porque não é composição
43 end;

44 procedure Button.Display();
45 begin

46   WriteLn(_name);
47 end;

48 procedure Button.Remove(c: Component);
49 begin
50   // não remove porque não é composição
51 end;

52 { Label }

53 procedure &Label.Add(c: Component);
54 begin
55   // não adiciona porque não é composição
56 end;

57 procedure &Label.Display();
58 begin
59   WriteLn(_name);
60 end;

61 procedure &Label.Remove(c: Component);
62 begin
63   // não remove porque não é composição
64 end;

65 { Panel }

66 procedure Panel.Add(c: Component);
67 begin
68   // não adiciona porque não é composição
69 end;

70 procedure Panel.Display();
71 begin
72   WriteLn(_name);
73 end;

74 procedure Panel.Remove(c: Component);
75 begin
76   // não remove porque não é composição
77 end;

78 { Image }

79 procedure Image.Add(c: Component);
80 begin
81   // não adiciona porque não é composição
82 end;

83 procedure Image.Display();
84 begin
85   WriteLn(_name);
86 end;

87 procedure Image.Remove(c: Component);
88 begin
89   // não remove porque não é composição
90 end;

91 end.
```

Para uma melhor visualização, a **Listagem 8** ilustra as implementações dos métodos da composição (formulário) e das Leafs (demais controles).

A **Listagem 9** mostra como simular a exibição de um formulário formado pela composição de outros controles. Observe que o efeito é semelhante à criação de componentes em Runtime utilizando a própria VCL do Delphi, porém, fazendo uso do método Add.

Listagem 9. Usando um Composite

```
01 program Composite;

02 {$APPTYPE CONSOLE}

03 {$R *.res}

04 uses
05   System.SysUtils,
06   uFramework in 'uFramework.pas';

07 var
08   form: Formulario;
09 begin

10   form := Formulario.Create('Padrões de Projeto em Delphi');
11   form.Add(Panel.Create(''));
12   form.Add(&Label.Create('Cadastro de Clientes'));
13   form.Add(TextBox.Create('Código'));
14   form.Add(TextBox.Create('Nome'));
15   form.Add(TextBox.Create('Fone'));
16   form.Add(TextBox.Create('Endereço'));
17   form.Add(Button.Create('Incluir'));
18   form.Add(Button.Create('Consultar'));
19   form.Add(Image.Create('Logo'));
20   // poderia chamar pois form é tanto composição quanto componente
21   //form.Add(UmOutroForm);
22   form.Display();
23   ReadLn;
24 end.
```

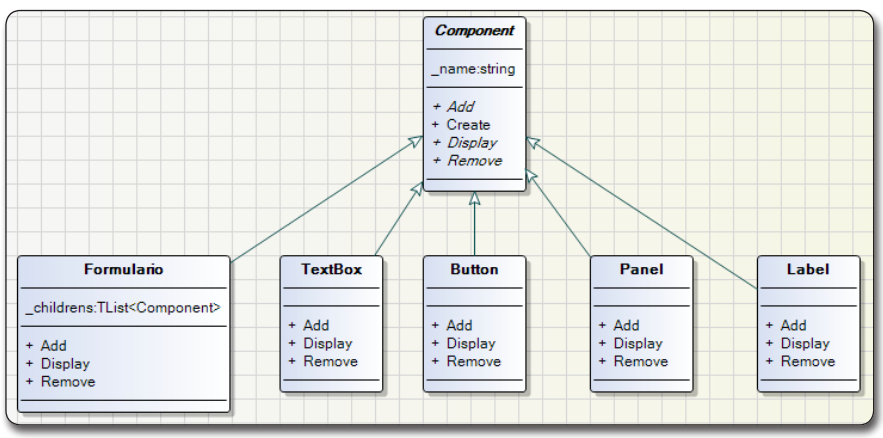

Figura 6. Estrutura de classes do exemplo usando Composite

Decorator

Este padrão tem a intenção de, dinamicamente, agregar responsabilidades adicionais a um objeto. Os decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.

O padrão, conhecido também como wrapper, favorece o uso de composição sobre a herança. Permite adicionar (ou remover) responsabilidades individuais a uma classe de forma dinâmica, flexível e transparente. Ao usar herança, uma classe formalmente especifica seu tipo herdado e gera uma relação de dependência praticamente imutável, gerando fortes acoplamentos. O padrão Decorator, ao invés disso, permite estender uma classe "decorando-a" com novas funcionalidades, sem que seja necessário criar uma grande hierarquia de subclasses. Os benefícios do uso do padrão são inúmeros, como principal e já citado é a flexibilidade, pois não faz uso de herança, que é estática. Além disso, evita criar uma classe com inúmeras responsabilidades expostas em sua interface, já que a "decoração" é interna. Funções são delegadas internamente sem exposição pública.

Os participantes do padrão de projeto são:

- Component – define a interface para objetos que podem ter responsabilidades acrescentadas dinamicamente;
- ConcreteComponent – define um objeto para o qual as responsabilidades adicionais podem ser atribuídas;
- Decorator – mantém uma referência para um objeto Component e define uma interface que segue a interface de Component;
- ConcreteDecorator – acrescenta responsabilidades ao componente.

A **Listagem 10** mostra o funcionamento de um framework de classes que implementa o padrão Decorator. Seu diagrama de classes pode ser visto na **Figura 7**.

Embora a **Listagem 10** seja extensa, seu conteúdo é simples. Definimos uma interface para um determinado componente (linhas 05 a 08), bem como sua implementação concreta (linhas 10 a 14). Ao invés de adicionarmos inúmeras operações neste objeto concreto, espalhamos suas demais operações por várias classes de decoração. Cada Decorator possui uma referência para uma abstração do componente (linha 18), então sabe qual classe deve ser decorada com novas funcionalidades. A **Listagem 11** mostra como usar esse framework. Criamos um componente (linha 15) e a seguir vários Decorators (linhas 17 a 20). Através do método SetComponent declarado no Decorator, conseguimos "decorar" um determinado componente e adicionar funcionalidades a ele sem modificá-lo (linhas 22 a 25), nem mesmo sua interface.

Façade

A intenção do padrão Façade é fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais elevado que faz o subsistema mais

Listagem 10. Decorator

```
01 unit uFramework;

02 interface

03 type
04  // "Component"
05 Component = class abstract
06 public
07   procedure Operation(); virtual; abstract;
08 end;

09 // "ConcreteComponent"
10 ConcreteComponent = class (Component)
11 public
12   procedure Operation(); override;
13  // + métodos
14 end;

15 // "Decorator"
16 Decorator = class (Component)
17 protected
18   _component: Component;
19 public
20   procedure SetComponent(aComponent: Component);
21  // método adicional você normalmente adicionaria
22  // diretamente no concrete componente
23   procedure Operation(); override;
24 end;

25 // "ConcreteDecoratorA"
26 ConcreteDecoratorA = class(Decorator)
27 private
28   addedState: string;
29 public
30   procedure Operation(); override;
31 end;

32 // "ConcreteDecoratorB"
33 ConcreteDecoratorB = class(Decorator)
34 private
35   addedState: string;
36 public
37   procedure Operation(); override;
38 end;

39 // "ConcreteDecoratorC"
40 ConcreteDecoratorC = class(Decorator)
41 private
42   addedState: string;
43 public
44   procedure Operation(); override;
45 end;

46 // "ConcreteDecoratorD"
47 ConcreteDecoratorD = class(Decorator)
48 private
49   addedState: string;
50 public
51   procedure Operation(); override;
52 end;

53 implementation

54 { ConcreteComponent }

55 procedure ConcreteComponent.Operation();
56 begin
57   WriteLn('ConcreteComponent.Operation');
58 end;

59 { Decorator }

60 procedure Decorator.Operation();
61 begin
62   if (_component <> nil) then
63     _component.Operation();
64  // operação concreta...
65  // + operação decorada, adicionada dinamicamente
66 end;

67 procedure Decorator.SetComponent(aComponent: Component);
68 begin
69   self._component := aComponent;
70 end;

71 { ConcreteDecoratorA }

72 procedure ConcreteDecoratorA.Operation();
73 begin
74   inherited Operation();
75   addedState := 'New State';
76   WriteLn('ConcreteDecoratorA.Operation');
77 end;

78 { ConcreteDecoratorB }

79 procedure ConcreteDecoratorB.Operation();
80 begin
81   inherited Operation();
82   addedState := 'New State';
83   WriteLn('ConcreteDecoratorB.Operation');
84 end;

85 { ConcreteDecoratorC }

86 procedure ConcreteDecoratorC.Operation();
87 begin
88   inherited Operation();
89   addedState := 'New State';
90   WriteLn('ConcreteDecoratorC.Operation');
91 end;

92 { ConcreteDecoratorD }

93 procedure ConcreteDecoratorD.Operation();
94 begin
95   inherited Operation();
96   addedState := 'New State';
97   WriteLn('ConcreteDecoratorD.Operation');
98 end;

99 end.
```

Listagem 11. Usando um Decorator

```
01 program Decorator;

02 {$APPTYPE CONSOLE}

03 {$R *.res}

04 uses
05   System.SysUtils,
06   uFramework in 'uFramework.pas';

07 var
08   c: ConcreteComponent;
09   d1: ConcreteDecoratorA;
10   d2: ConcreteDecoratorB;
11   d3: ConcreteDecoratorC;
12   d4: ConcreteDecoratorD;
13 begin
14   // Cria Concrete Component (componente concreto)
15   c := ConcreteComponent.Create();
16   // Cria 3 Decorators
17   d1 := ConcreteDecoratorA.Create();
18   d2 := ConcreteDecoratorB.Create();
19   d3 := ConcreteDecoratorC.Create();
20   d4 := ConcreteDecoratorD.Create();
21   // Linka decorators com component concreto
22   d1.SetComponent(c);
23   d2.SetComponent(c);
24   d3.SetComponent(c);
25   d4.SetComponent(c);
26   // chama método decorados que invocam
27   // método base de component
28   // injetando código em componente sem modificar
29   d1.Operation();
30   d2.Operation();
31   d3.Operation();
32   d4.Operation();
33   Readln;
34 end.
```

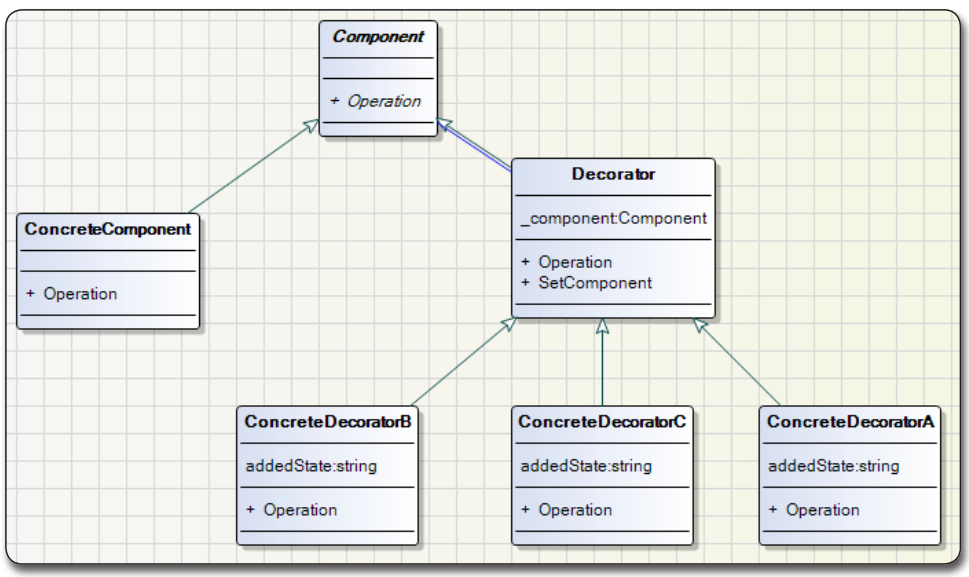

Figura 7. Estrutura de classes do exemplo usando Decorator

fácil de usar. Uma boa aplicação para o Façade (fachada) é criar uma forma mais simples para os objetos clientes de lideram com um sistema mais complexo, escondendo muito de sua implementação, fortalecendo o encapsulamento e isolamento. Alguns benefícios do padrão: tornar uma biblioteca de software mais fácil de entender e usar; tornar o código que utiliza esta biblioteca mais fácil de entender; reduzir as dependências em relação às características internas de uma biblioteca, trazendo flexibilidade no desenvolvimento do sistema; envolver uma interface mal desenhada, com uma interface melhor definida. Os participantes do padrão de projeto são:

Façade – conhece quais as classes do subsistema são responsáveis pelo atendimento de uma solicitação; delega solicitações de clientes a objetos apropriados do subsistema;

Classes de subsistema – implementam as funcionalidades do subsistema; encarregam-se do trabalho atribuído a elas pelo objeto Façade; não tem conhecimento (referência) ao Façade;

Para entender melhor como funciona o padrão Façade, novamente vamos fazer comparações com objetos do mundo real. Vamos comprar um novo apartamento, o mesmo possui uma boa fachada, bonita externamente. Uma fachada muito bonita muitas vezes pode estar servindo para esconder algo que não está muito bem por dentro, ou pelo menos, não deveria ser visível ou de conhecimento do público. A fachada dessa forma serve para esconder algo duvidoso, e seu novo apartamento pode trazer problemas. Quando algo está muito ruim, difícil de entender ou complexo, normalmente se dá uma interface mais amigável, sem modificar seu interior, para causar uma boa aparência, ou facilitar sua visualização e utilização.

No mundo orientado a objetos, muitos frameworks são tão bem projetados e seguem tão bem à risca o Princípio da Responsabilidade Única que é preciso instanciar quatro classes para realizar uma simples tarefa. Por exemplo, no ADO.NET com SQL Server, para executar um comando SQL de seleção (select) no banco de dados e guardar isso em cache, é necessário as classes: SqlConnection, SqlCommand, SqlDataAdapter e DataSet. De forma muito semelhante ocorre no DBExpress. É muito comum nesses casos criar classes ou funções "Helpers", que facilitam a vida do desenvolvedor ao utilizar todo esse conjunto de classes. Por exemplo, poderíamos ter uma função chamada ExecutaSQL(SQL: string), que configura cada uma dessas classes, relaciona-as e executa um comando, passando parâmetros e retornando um resultset caso seja um select. Nesse caso, temos uma função ou classe que esconde um conjunto de operações mais complexas. Ou seja, é uma fachada para esconder algo que está complexo de utilizar. Outras vezes, um subsistema

de classes está tão mal projetado e difícil de utilizar que só uma fachada resolve: esconde-se tudo o que tem até então e a partir daí os programadores só usam a fachada. Agora paremos para pensar: O que realmente é a VCL? Não é um conjunto de classes que permite usar um subsistema de funções extremamente complexo e sem classe / tipo, que é a API procedural do Windows? Sim, certamente, é muito simples criar um TForm via VCL do que via API do Windows. A própria plataforma .NET Framework é uma grande fachada para o sistema Windows, escondendo sua complexidade de forma a oferecer um conjunto de classes mais fáceis de trabalhar, porém, com as mesmas funcionalidades que o sistema operacional oferece de forma nativa. Talvez estes sejam os exemplos mais reais atualmente de aplicação do padrão de projeto Façade.

A **Listagem 12** mostra um exemplo de uso do padrão Façade, as classes declaradas na unit podem ser vistas na **Figura 8.**

Temos várias classes destinadas ao envio de mensagens de e-mail. Um e-mail tem um formato (HTML ou TXT – linhas 06 a 09), é enviado através de um servidor SMTP (linhas 18 a 27), quem tem suas configurações. A mensagem é representada por um tipo chamado MailMessage (linha 41), ao invés de uma string primitiva. Agora, se for necessário enviar um simples e-mail com um

Listagem 12. Façade

```
01 unit uFramework;

02 interface

03 type

04  MailFormat = class

05  end;

06  HTMLFormat = class(MailFormat)
07  end;

08  TXTFormat = class(MailFormat)
09  end;

10  MailMessage = class
11  private
12    _mensagem: string;
13    _format: MailFormat;
14  public
15    property Mensagem: string read _mensagem write _mensagem;
16    constructor Create(mensagem: string; format: MailFormat);
17  end;

18  SmtpSettings = class
19  private
20    _serverName: string;
21    _userName: string;
22    _password: string;
23  public
24    property ServerName: string read _serverName write _serverName;
25    property UserName: string read _userName write _userName;
26    property Password: string read _password write _password;
27  end;

28  Mail = class
29  private
30    _conf: SMTPSettings;
31  public
32    constructor Create(conf: SMTPSettings);
33    procedure Send(Mensagem: MailMessage);
34  end;

35  // Façade
36  Email = class
37  private
38    _mail: Mail;
39    _format: MailFormat;
40    _conf: SMTPSettings;
41    _msg: MailMessage;
42  public
43    constructor Create();
44    procedure Enviar(aMensagem: string);
45  end;

46 implementation

47 constructor MailMessage.Create(mensagem: string; format: MailFormat);
48 begin
49  // formatar mensagem aqui...
50  self._mensagem := mensagem;
51  self._format := format;
52 end;

53 { Mail }

54 constructor Mail.Create(conf: SMTPSettings);
55 begin
56  self._conf := conf;
57 end;

58 procedure Mail.Send(Mensagem: MailMessage);
59 begin
60  // usando configurações SMTP
61  WriteLn('Enviando mensagem...');
62  WriteLn(Mensagem.Mensagem);
63 end;

64 { Email }

65 constructor Email.Create();
66 begin
67  self._conf := SMTPSettings.Create();
68  self._conf.ServerName := 'smtp.gmail.com';
69  self._conf.Password := '123';
70  self._conf.UserName := 'admin';
71  self._mail := Mail.Create(_conf);
72  self._format := TXTFormat.Create();
73  self._msg := MailMessage.Create('',_format);
74 end;

75 procedure Email.Enviar(aMensagem: string);
76 begin
77  _msg.Mensagem := AMensagem;
78  _mail.Send(_msg);
79 end;

80 end.
```

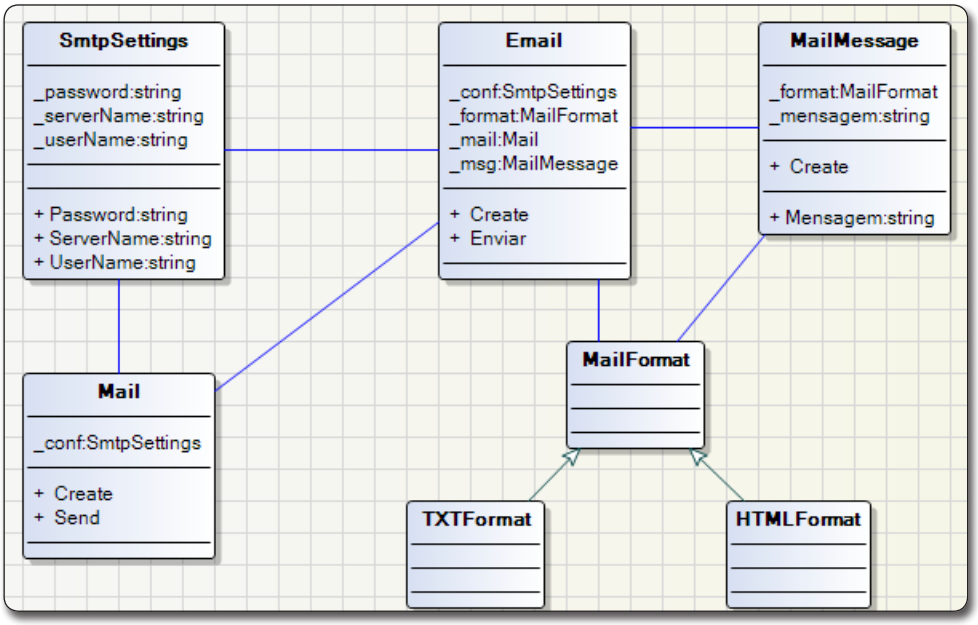

Figura 8. Estrutura de classes do exemplo usando Façade

Listagem 13. Usando um Façade

```
01 program Facade;

02 {$APPTYPE CONSOLE}

03 {$R *.res}

04 uses
05   System.SysUtils,
06   uFramework in 'uFramework.pas';

07 var
08   em: Email;
09 begin
10   em := Email.Create();
11   em.Enviar('Olá Mundo');
12   Readln;
13 end.
```

"Olá Mundo", precisaremos instanciar e configurar todas essas classes, definir propriedades e chamar os respectivos métodos. Para complicar, algumas configurações são passadas no construtor de outras classes. E como saber o que realmente deve ser configurado para enviar o e-mail? Quais propriedades são opcionais e quais obrigatórias?

Quais métodos devem ser chamados, e em que ordem? Note no final do framework que foi criada uma classe chamada "Email", ela é o Façade (linhas 36 a 41), e esconde toda essa complexidade. Basta criar uma instância desta classe e chamar seu método Enviar, como mostra a **Listagem 13**.

Conclusão

Neste artigo vimos como implementar alguns padrões estruturais GoF (Gang Of Four) em Delphi, incluindo o Adapter, Bridge, Composite, Decorator e Façade. Tais padrões, se bem empregados, permitem otimizar a reutilização de código, reduzir o acoplamento entre partes da aplicação, melhorando e facilitando a manutenção. Como vimos, o próprio Delphi se apoia em padrões de projeto famosos para constituir seu IDE e a VCL / FMX.

Autor

Guinther Pauli
http://www.facebook.com/guintherpauli
http://www.twitter.com/guintherpauli
http://br.linkedin.com/in/guintherpauli
http://www.gpauli.com
http://guintherpauli.blogspot.com.br
http://www.devmedia.com.br/guintherpauli
Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C# e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix. Most Valuable Professional (MVP) - Embarcadero Technologies. Entre os principais projetos que participou, foi arquiteto e desenvolvedor do sistema de doações on-line do projeto Criança Esperança da Rede Globo / Unesco. Beta Test de várias versões do Delphi, palestrante em 5 edições da Borland Conference, maior evento de desenvolvedores do país. Ministra treinamentos corporativos in-company, já tendo treinado profissionais de várias empresas dos estados do RS, SC, PR, SP, RJ, GO, TO, DF, MG, AM e PE. Autor de mais de 1100 videoaulas de ensino a distância pela DevMedia, publicou mais de 130 artigos, além do livro Delphi Programação para Banco de Dados e Web . Fez a cobertura de grandes eventos no exterior, como o TechEd da Microsoft e a BorCon da Borland americana. Experiência em Delphi de 18 anos e 13 anos em .NET / C#.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Links:

GoF Patterns – Quick Review
http://www.mydeveloperconnection.com/html/gof_design_patterns.htm

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8.000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 1,20

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Uso de Voz em Delphi

Aprenda a ouvir e executar comandos

Em 11 de agosto de 2001, a Microsoft libera o SAPI 5.1 SDK na qual permite seu uso em qualquer linguagem que suporte automação OLE (**BOX 1**). SAPI é o acrônimo de Speech Application Programming Interface. Consiste em uma API desenvolvida pela Microsoft na qual possibilita o reconhecimento de voz em aplicações Windows. Um exemplo conhecido de software que faz uso desta API é o Microsoft Office.

Os componentes do SAPI SDK podem ser utilizados para elaborar sistemas com reconhecimento de voz e leitura de dados. Essa tecnologia que é muito robusta pode ser usada não só em Delphi, como em diversas outras linguagens de programação.

BOX 1. OLE Automation

OLE Automation ou apenas Automation é um mecanismo comumente utilizado para realizar a comunicação entre aplicações baseadas em COM (Component Object Model). A OLE permite o uso e manipulação de aplicações, ou seja, uma aplicação controlando outra.

Resumo DevMan

Uso de Voz em Delphi:

Sistemas de empresas telefônicas e muitas outras hoje em dia, usam recursos de voz devido a simplicidade gerada para entrada de dados. O mesmo poderia ser aplicado nos mais diversos sistemas, por exemplo, em consulta de clientes, agendamentos entre outras situações. É mais fácil e muito mais ágil informar os dados através da voz, no lugar do uso do mouse ou teclado. Obviamente em algumas outras situações, é mais fácil ouvir as informações no lugar de ler. É justamente o que abordaremos neste artigo, fazendo com que o nosso exemplo leia dados dos clientes previamente cadastrados por meio de comandos de voz.

Em que situação o tema é útil:

O tema é útil em casos onde há a necessidade de criação de consultas para agilizar processos, além disso, também pode ser utilizado como forma de auxiliar usuários que sofram de algum tipo de deficiência ou dificuldade em manusear o mouse ou teclado.

Usando a tecnologia do SDK SAPI 5.1

Para seu uso em Delphi, é necessário instalar o SDK SAPI 5.1 e configurá-lo, sendo apenas um detalhe mínimo, bastando importar sua Type Library e usar conforme a necessidade.

Em aplicações que irão usar leitura de textos, não é necessário ter nenhum reconhecimento de voz (SDK) instalado na máquina do cliente, sendo assim, limita-se apenas ao uso das "vozes" que estão instaladas no próprio Windows.

Além disso, podem ser instaladas vozes adicionais para o uso da aplicação, estas são identificadas por nomes de pessoas como as de "Raquel" (Português) e "Alonso" (Espanhol). No caso do idioma em português, existem outras opções, no entanto, a voz mais completa e de entonação melhor seria a voz denominada como "Raquel". Na seção **Links** foi disponibilizado seu download.

Para usar os recursos de reconhecimento de voz, deve-se obrigatoriamente atender alguns pré-requisitos. Sendo assim, devem ser instalados no Windows, dentro de Painel de Controle, o reconhecimento de voz, assim como um microfone e a parte de som, que deverá estar funcionando corretamente.

Leitura de textos

Para leitura de textos, não é necessário ter nenhum reconhecimento de voz instalado. O computador irá apenas reproduzir o texto, usando os fonemas do idioma da voz selecionada para a "leitura" do texto.

Reconhecimento de comandos simples

Para um uso de reconhecimento de comandos simples, pra controlar botões e controles diversos dos formulários em geral, basta criar um objeto de reconhecimento (SpSharedRecoContext) e uma gramática para o mesmo. Esta gramática é o conjunto de palavras (comandos), que serão reconhecidos pela aplicação e que podem ser carregados por meio de um arquivo, banco de dados, ou um XML que é o mais comum. Desta forma os usuários podem falar o que for, que será descartado pelo objeto de reconhecimento. Quando uma palavra for falada e estiver na gramática do reconhecimento (XML, por exemplo) o aplicativo irá executar a ação correspondente do seu código.

Reconhecimento exato

Para um reconhecimento exato, é interessante que os comandos sejam palavras curtas e de pronúncias bem distintas uma das outras, assim não haverá equívocos.

Em nosso exemplo adotaremos o menu de uma aplicação qualquer, com os itens "novo", "abrir" e "sair".

Na gramática do reconhecimento é interessante que os itens estejam exatamente descritos como "novo", "abrir" e "fechar", para não causar nenhuma falha entre "abrir" e "sair", por exemplo, devido a semelhança, voz, entonação, velocidade da fala e etc.

Gramática de reconhecimento em XML

Para executar os itens de um menu, como o mencionado anteriormente, nosso objeto de reconhecimento (SpSharedRecoContext) seria carregado com uma estrutura semelhante ao arquivo XML definido na **Listagem 1**.

Listagem 1. Arquivo XML com a definição de reconhecimento para os valores novo, abrir e fechar

```
01 <?xml version="1.0"?>
02 -<GRAMMAR LANGID="809">
03     <!-- "Constant" definitions -->
04   -<DEFINE>
05     <ID VAL="1" NAME="RID_start"/>
06     <ID VAL="2" NAME="PID_cmdesolhido"/>
07     <ID VAL="3" NAME="PID_cmdvalor"/>
08   </DEFINE>
09     <!-- Rule definitions -->
10 -<RULE NAME="start" TOPLEVEL="ACTIVE" ID="RID_start">
11    <O>cmd</O>
12    <RULEREF NAME="cmd" PROPID="PID_cmdesolhido"
        PROPNAME="cmdesolhido"/>
13    </RULE>
14   -<RULE NAME="cmd">
15     -<L PROPID="PID_cmdvalor" PROPNAME="cmdvalor">
16         <P VAL="1">novo</P>
17         <P VAL="2">abrir</P>
18         <P VAL="3">fechar</P>
19      </L>
20    </RULE>
21 </GRAMMAR>
```

Uma prática que poderia ser adotada seria ao chamar uma tela de cadastro ou movimentações, por exemplo, criar dinamicamente esse XML com os comandos da tela a partir de um banco de dados, onde os atributos (VAL="1") seriam os códigos da tabela e os valores ("NOVO") seriam os campos do comandos de voz. Observe que a descrição dos itens encontram-se descritos códigos apresentados entre as linhas 16 a 18 da **Listagem 1**, indicados pelas tags VAL.

Conhecendo o objeto (SpVoice)

Veremos a seguir as principais propriedades do objeto SpVoice, onde algumas situações será necessário fazer uma ou mais mudanças para melhor se adequar a nossa necessidade:

• Priority - A propriedade "Priority" obtém e define o nível de prioridade da voz. O nível de prioridade define a ordem em que o mecanismo de texto processa os pedidos de fala de um objeto de voz (SpVoice). Níveis de prioridade mais elevados são atribuídos às vozes de tratamento de erros, já menores níveis de prioridade são atribuídos às vozes normais. Por causa de seu nível de prioridade, os pedidos de vozes de tratamento de erros (geralmente usados para falar as mensagens de erros), são falados antes de outras solicitações normais de voz. As vozes de tratamento de erros podem aparecer para interromper as vozes normais. Um objeto SpVoice, por padrão, é criado com a prioridade normal. Para utilizar uma ou mais vozes de alerta, basta criá-la e definir sua propriedade "Priority" adequadamente. Uma voz com uma definição de prioridades do tipo SVPAlert é tratada como uma voz de alerta. Vozes de alerta são projetadas para serem o principal meio para tratamento de erros.

• Rate - É responsável por obter e definir a velocidade da fala da voz. É definida por intervalos de -10 a 10, que representam do mais lento ao mais rápido respectivamente. No início de cada método de fala, a voz define a sua velocidade de leitura de acordo com o valor das suas propriedades Rate e pronuncia todo o fluxo com essa taxa. Essa propriedade pode ser alterada a qualquer momento, porém, a velocidade da fala real não irá refletir o valor da propriedade atual até que se inicie um novo fluxo.

• Voice - Obtém e define a voz ativa da coleção de vozes. Esta propriedade pode ser pensada como a pessoa que irá pronunciar os textos, alguns exemplos de vozes são denominados de "Microsoft Mary" e "Microsoft Mike". Para saber as vozes que estão disponíveis, basta usar o método "GetVoices". Se já não houver uma voz em uso, essa propriedade apontará para a padrão definida no Microsoft Windows.

• Volume - A propriedade "Volume" obtém e define o volume base do nível da voz. Valores entre o intervalo de 0 a 100 representam os níveis mínimo e máximo de volume respectivamente. No início de cada método de fala, a voz define o volume de acordo com o valor desta propriedade e pronuncia todo o texto corrente nesse nível. Esta propriedade pode ser alterada a qualquer momento, porém, o nível de volume atual não reflete o valor alterado até que se inicie um novo fluxo.

Eventos

Veremos a seguir os principais eventos que contém um objeto SpVoice. A fim de compreender os eventos de voz, é necessário fazer a distinção entre o mecanismo de TTS que sintetiza a fala do texto e o objeto SpVoice que se comunica com o mecanismo de TTS.

O mecanismo TTS funciona como se fosse um servidor e o objeto SpVoice como um cliente. O objeto de voz envia ao TTS um pedido para falar uma sequência de texto, por sua vez, o TTS processa o pedido. O intervalo de tempo entre um pedido e a produção do discurso é imprevisível. Os eventos do SpVoice podem superar esta dificuldade, fornecendo meios de obter respostas em tempo real do TTS, o que torna possível sincronizar funções do aplicativo com o discurso.

O objeto de voz envia os pedidos com os métodos Speak e SpeakStream. Esses enviam cadeias de texto e arquivos de áudio para o TTS. Estes métodos podem ser chamados de forma síncrona ou assíncrona. Com um pedido de fala chamado de forma síncrona, a execução do aplicativo é suspensa enquanto o texto é falado e os eventos da fala do texto são recebidos depois que o fluxo foi falado. Esses são os eventos disponíveis:

• AudioLevel - Ocorre quando o mecanismo de texto para fala (TTS) detecta uma mudança de nível de áudio enquanto estiver enviando um fluxo para o objeto SpVoice.
• Bookmark - O evento Bookmark ocorre quando o TTS detecta um marcador enquanto estiver enviando um fluxo para o objeto SpVoice. Deve-se observar que o evento marcador pode não estar sincronizado com a palavra do texto falado, ou seja, pode não estar marcando a palavra correta do texto. Em algumas circunstâncias, o TTS pode marcar mais cedo a palavra do fluxo do texto.
• Sentence – É disparado toda a vez que o TTS detecta uma sentença na frase ao enviar um fluxo para o objeto SpVoice.
• StartStream e EndStream - StartStream ocorre quando o mecanismo de texto para fala começa a enviar um fluxo para o objeto SpVoice. Já o evento EndStream ocorre quando se dá o fim do fluxo. Esses eventos podem ser usados em conjunto para determinar a duração de um fluxo de texto a ser pronunciado.
• Phoneme - O evento Phoneme ocorre quando é detectado um fonema ao enviar um fluxo para o objeto SpVoice.
• Viseme - É um gatilho para cada reconhecimento. Ele requer imagens para representar os fonemas identificados no discurso. Um fonema é a menor unidade de uma linguagem que pode transmitir um significado, como o som "M" em Maria. Cada fonema tem sua respectiva forma facial. Animadores normalmente associam fonemas com as formas que a boca toma ao criar vários sons. Estas formas da boca por sua vez, são conhecidas tecnicamente como visemes. Animadores da Disney por exemplo, se basearam em um gráfico de 13 posições arquetípicas da boca para representar a fala do idioma inglês. Com uma certa aptidão artística fica fácil desenhar os visemes necessários para a representação de qualquer idioma, usando imagens de rosto de uma pessoa, ou de um robô, como é visto hoje em dia em vários filmes.
• VoiceChange - O evento VoiceChange ocorre quando o detecta uma mudança de voz ao enviar um fluxo para o objeto SpVoice.
• Word - O evento Word ocorre quando o mecanismo detecta uma nova palavra ao enviar um fluxo para o objeto SpVoice.

Métodos do objeto SpVoice

A seguir conheceremos alguns dos principais métodos do SpVoice. São eles:
•GetVoices - É responsável por retornar as vozes disponíveis no Microsoft Windows para o objeto SpVoice. Os parâmetros de busca podem ser aplicados opcionalmente. Na ausência de parâmetros de busca, todas as vozes são devolvidas para o objeto em ordem

CURSOS ONLINE

A Revista .net Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

.net magazine

CONHEÇA OS CURSOS MAIS RECENTES:

• Curso Padrões de Projeto com C#
• Curso básico de ASP .NET
• Curso de Introdução ao .NET Framework
• Curso Básico de C#
• C# 5 e suas novidades
• ASP.NET MVC – Sistema de Vestibular

Para mais informações :
www.devmedia.com.br/curso/netmagazine
(21) 3382-5038

DevMedia

alfabética pelo nome da voz. Se não houver vozes que correspondem aos parâmetros de busca, o retorno do método é vazio.

• Speak - **O método Speak é para a fala do texto. Pode ser chamado** de forma síncrona ou assíncrona. Quando chamado de forma síncrona, o método não retornará até que o texto seja falado, quando chamado de forma assíncrona, ele retorna imediatamente e a voz é pronunciada como um processo de fundo. Quando o método síncrono é utilizado em uma aplicação, a execução do aplicativo é bloqueada enquanto a voz fala, não permitindo o controle do usuário. Isso pode ser aceitável para aplicações simples, ou aquelas sem interface gráfica do usuário (GUI), mas, quando a interação do usuário enquanto o texto é falado se faz necessária, o modo assíncrono é mais apropriado. O método Speak insere um fluxo na fila do TTS e retorna um número atribuído pelo mecanismo. Isso distingue esse fluxo de outros que também estão na fila. Este número é um identificador temporário que funciona como um índice para a fila do TTS. Um objeto de voz (SPVoice) pode enfileirar vários fluxos, e cada um desses fluxos podem gerar eventos variados, sendo os eventos associados com o número do fluxo.

• Pause - **É responsável por pausar a voz do objeto em uso, fe**chando o dispositivo de saída, permitindo que seja utilizado por outros objetos de voz.

• Resume - Faz com que a voz do objeto que foi pausada continue falando.

• Skip - Ignora a voz pelo número especificado de itens dentro do fluxo atual do texto. Funciona como se fosse um botão stop de um player, onde o número de itens informado no parâmetro é o que será ignorado pelo objeto. Se desejar parar totalmente a fala, basta informar o parâmetro "MaxInt" para que o fluxo que estiver sendo falado pelo objeto seja interrompido.

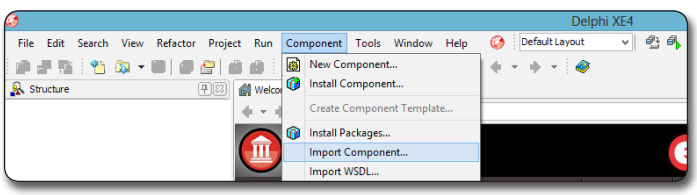

Figura 1. Importando componente ActiveX

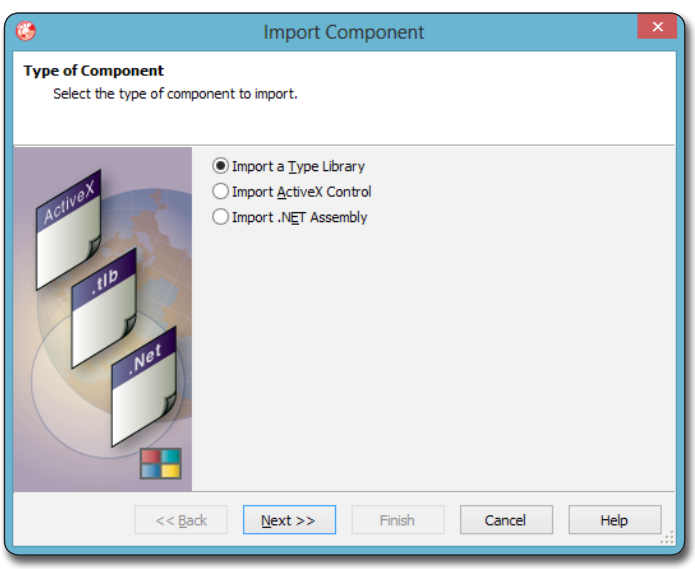

Figura 2. Opção para importar Type Library

Instalando o componente

Para a instalação do ActiveX no Delphi, basta acessar o menu Component>Install Component (**Figura 1**). Na tela que se abre (**Figura 2**), selecione a opção "Import a Type Library". Após o processo, selecione o componente dentre os exibidos na lista, conforme a **Figura 3**. Após a seleção, será adicionada a unit SpeechLib_TLB.pas (**Figuras 4** e **5**) com os novos componentes do SAPI 5.1. Utilizando a paleta de componentes padrão, sua instalação estará na aba ActiveX.

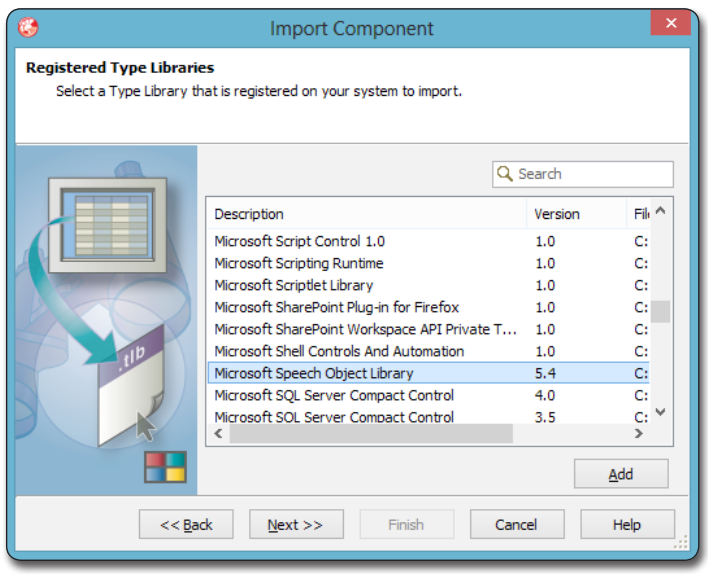

Figura 3. Seleção das bibliotecas disponíveis.

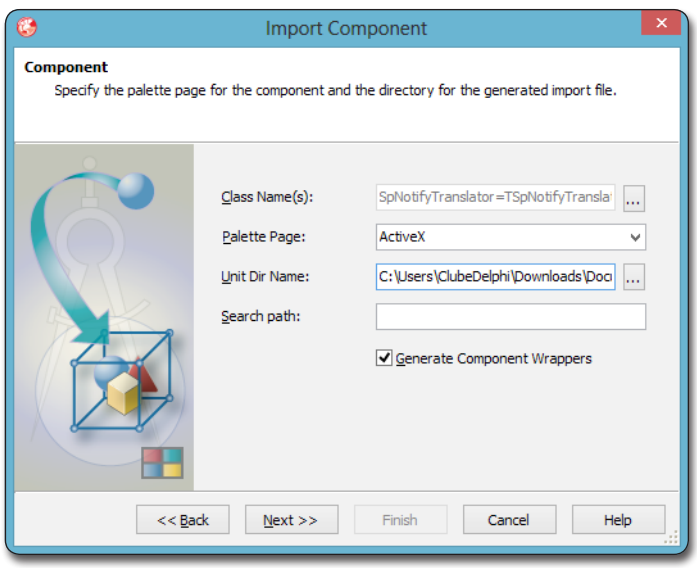

Figura 4. Setando a aba de instalação do componente

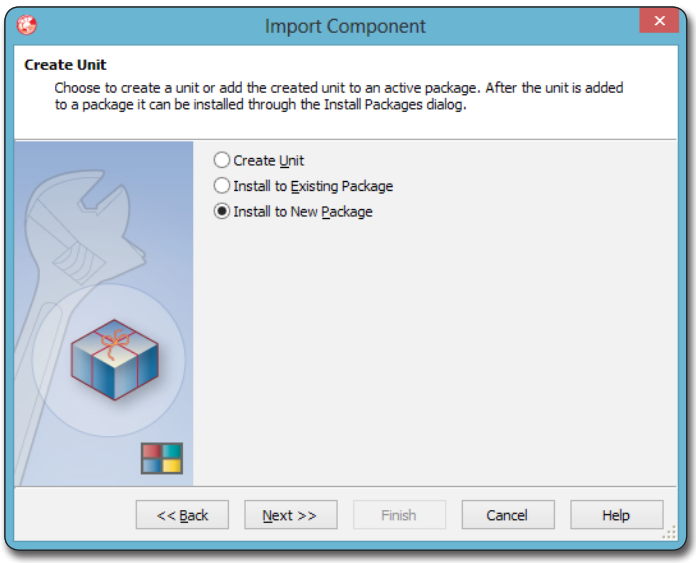

Figura 5. Última opção do Wizard de instalação

Criando a aplicação

Após a instalação do componente, é criada uma aplicação do tipo VCL Application. Para isso, selecionamos o menu New>VCL Application – Delphi.

Renomeamos o formulário principal para Frm_Principal e adicionamos alguns componentes nomeando-os conforme a seguir: três Labels (Lbl_Voz, Lbl_Pos e Lbl_Veloc), um TrackBar (tbRate), um Button (Btn_Ouvir), um ComboBox, (Cbx_Voz), um DBGrid, (Dbg_Cadastro), um RichEdit (Rch_Obs), um DBNavigator (Nvg_Cadastro), um ClientDataSet (Cds_Cadastro), um DataSource, (Ds_Cadastro) e por último um SpVoice, (SpVoice). O layout do formulário é ajustado conforme a **Figura 6.** Para melhor identificação na montagem da tela os Labels (Lbl_Voz, Lbl_Veloc, Lbl_Pos) possuem os respectivos Captions "Selecione a Voz", "Velocidade da Voz", "00".

Para criar a comunicação com os dados e fazer a codificação necessária, adicionamos três fields no ClientDataSet dando um duplo clique sobre o mesmo e selecionando a opção New Field. Os campos a serem incluídos encontram-se na **Tabela 1**.

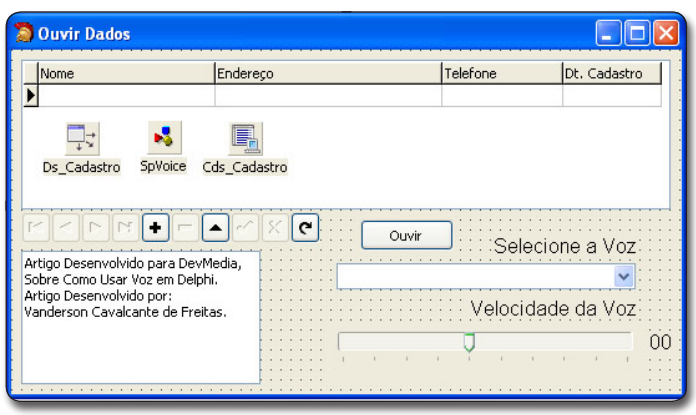

Figura 6. Tela Principal de cadastro

Nota

Foram adicionadas as units OleServer e SpeechLib_TLB ao uses do formulário.

Nome do campo	Tipo de dado	Tamanho
Nome	String	20
Endereço	String	50
Telefone	String	15
DtCadastro	Date	-

Tabela 1. Definição dos campos do ClientDataSet

Inseridos os campos, basta ativar o ClientDataSet usando o método CreateDataSet, fazendo com que este esteja preparado para a inserção de dados. Esta opção pode ser utilizada em Design Time, clicando de direita sobre o componente e selecionando-a no menu de contexto, ou em Runtime, por forma de comandos, utilizando o método CreateDataSet.

Ligamos o componente Ds_Cadastro ao ClientDataSet por meio de sua propriedade DataSet. Em seguida, informamos aos componentes Dbg_Cadastro e Nvg_Cadastro o DataSource configurado, apontando para a propriedade de mesmo nome. Assim a parte de visualização de dados já está configurada e podemos partir para a codificação.

No evento OnCreate do formulário vamos iniciar os componentes de voz e de dados, como mostra a **Listagem 2**.

Listagem 2. Evento OnCreate do formulário

```
01 procedure TFrm_Principal.FormCreate(Sender: TObject);
02 var
03  I: Integer;
04  SOTokens: ISpeechObjectTokens;
05  SOToken: ISpeechObjectToken;
06 begin
07  SpVoice.EventInterests := SVEAllEvents;
08  SOTokens          := SpVoice.GetVoices('','');
09  for I := 0 to SOTokens.Count - 1 do
10  begin
11   SOToken := SOTokens.Item(I);
12   Cbx_Voz.Items.AddObject(SOToken.GetDescription(0), TObject(SOToken));
13   SOToken._AddRef;
14  end;
15  if Cbx_Voz.Items.Count > 0 then
16  begin
17   Cbx_Voz.ItemIndex := 0;
18   Cbx_Voz.OnChange(Cbx_Voz);
19  end;
20  TbRate.Position  := SpVoice.Rate;
21  Lbl_Pos.Caption  := IntToStr(TbRate.Position);
22  with Cds_Cadastro do
23  begin
24   Insert;
25   FieldByName('NOME').AsString    := 'Angelina Jolie';
26   FieldByName('ENDERECO').AsString := 'Rua Gomes da Silva, Nº520';
27   FieldByName('TELEFONE').AsString := '(99)7218-3542';
28   FieldByName('DTCADASTRO').AsDateTime := Now;
29   Post;
30  end;
31 end;
```

Entre as linhas 03 a 05 são definidas as variáveis utilizadas no evento OnCreate. Vale salientar que SOTokens será responsável por obter uma lista de vozes instaladas, já o SOToken (no singular), tratará apenas uma única voz. Na linha 07 informamos ao componente que queremos visualizar todos os eventos disponíveis no componente. A linha 08 utiliza o método GetVoices para obter toda a lista de vozes instalada no computador. A seguir, na linha 09, é utilizado um laço para percorrer os itens encontrados. Observe então que a variável SOToken recebe a voz atual do laço (linha 11), adicionando-as ao ComboBox (linha 12). O próximo passo é verificar se há itens no ComboBox (linha 15), fazendo com que, caso seja verdadeiro, o primeiro item da lista seja selecionado (linha 17), acionando o evento OnChange do próprio componente (linha 18).

Entre as linhas 20 e 21 os componentes visuais são atualizados com os valores baseados na posição da barra e na propriedade Rate do SpVoice. Por último, entre as linhas 22 a 29, é simulada a inserção de um registro no ClientDataSet.

Após terminar o evento OnCreate, implementaremos o OnDestroy do formulário. A **Listagem 3** ilustra a codificação do evento.

O evento OnDestroy simplesmente percorre os itens do ComboBox (linha 05) e os libera da memória (linha 06). Continuando, implementamos o evento OnChange do ComboBox conforme ilustrado na **Listagem 4**.

Listagem 3. Evento OnDestroy do formulário

```
01 procedure TFrm_Principal.FormDestroy(Sender: TObject);
02 var
03   I: Integer;
04 begin
05   for I := 0 to Cbx_Voz.Items.Count - 1 do
06     ISpeechObjectToken(Pointer(Cbx_Voz.Items.Objects[I]))._Release;
07 end;
```

Listagem 4. Evento OnChange do ComboBox

```
01 procedure TFrm_Principal.Cbx_VozChange(Sender: TObject);
02 var
03   SOToken: ISpeechObjectToken;
04 begin
05   SOToken := ISpeechObjectToken(Pointer(
       Cbx_Voz.Items.Objects[Cbx_Voz.ItemIndex]));
06   SpVoice.Voice := SOToken;
07 end;
```

Podemos notar que simplesmente é obtido o objeto selecionado (a partir do ItemIndex do ComboBox -linha 05) e atribuído à variável SOToken. A seguir, esta variável é informada ao componente SpVoice (linha 06).

Na **Listagem 5** é implementado o evento OnChange do componente TbRate.

Neste evento fazemos com que caso a barra seja modificada, automaticamente a voz seja parada (linha 03), atribuindo ao Rate (velocidade), o valor selecionado no componente e atualizando o valor no Label (linhas 04 e 05 respectivamente).

Agora para finalizar, deverá ser implementado o evento OnClick do Button. O código encontra-se na **Listagem 6**.

Listagem 5. Evento OnChange da barra de velocidade (TbRate)

```
01 procedure TFrm_Principal.TbRateChange(Sender: TObject);
02 begin
03   SpVoice.Skip('Sentence', MaxInt);
04   SpVoice.Rate    := TbRate.Position;
05   Lbl_Pos.Caption := IntToStr(TbRate.Position);
06 end;
```

Listagem 6. Evento OnClick do botão Ouvir

```
01 procedure TFrm_Principal.Btn_OuvirClick(Sender: TObject);
02 begin
03   SpVoice.Skip('Sentence', MaxInt);
04   if (Cds_Cadastro.IsEmpty)then
05   begin
06     SpVoice.Speak('Não há dados cadastrado para leitura', SVSFlagsAsync);
07     Exit;
08   end;
09   if (Cds_CadastroNOME.AsString <> '') then
10   begin
11     SpVoice.Speak('Nome', SVSFlagsAsync);
12     SpVoice.Speak(Cds_CadastroNOME.AsString, SVSFlagsAsync);
13   end;
14   if (Cds_CadastroENDERECO.AsString <> '') then
15   begin
16     SpVoice.Speak('Endereço', SVSFlagsAsync);
17     SpVoice.Speak(Cds_CadastroENDERECO.AsString, SVSFlagsAsync);
18   end;
19   if (Cds_CadastroTELEFONE.AsString <> '') then
20   begin
21     SpVoice.Speak('Telefone', SVSFlagsAsync);
22     SpVoice.Speak(Cds_CadastroTELEFONE.AsString, SVSFlagsAsync);
23   end;
24   if (Cds_CadastroDTCADASTRO.AsString <> '') then
25   begin
26     SpVoice.Speak('Cadastrado', SVSFlagsAsync);
27     SpVoice.Speak(Cds_CadastroDTCADASTRO.AsString, SVSFlagsAsync);
28   end;
29   SpVoice.Speak(Rch_Obs.Text, SVSFlagsAsync);
30 end;
```

Assim como no anterior, interrompemos a voz caso esteja sendo falada alguma coisa (linha 03). Observe que caso não existam dados ao clicar no botão, o próprio erro será pronunciado pelo componente (linha 06). Entre as linhas a 09 a 29 os campos são validados um a um, tendo pronunciados seus Captions e seus respectivos conteúdos, juntamente com a observação do RichText ao final do código.

Conclusão

Neste artigo, conhecemos um pouco da tecnologia do SDK SAPI, que permite fazer uso de voz no Delphi. Este recurso é muito interessante para ler dados diversos nas mais variadas formas de utilidade. E claro, podemos ir mais além do que foi mostrado, fazendo uso do reconhecimento de voz em nas nossas telas de consultas, informando os dados com a voz por exemplo. Além disso, seria possível controlar menus, botões entre vários outros componentes simplesmente usando comandos de voz. Vale apena ler a documentação e explorar os recursos para integração com as aplicações.

Autor

Vanderson Cavalcante Freitas
vanderson.freitas@ig.com.br
Analista Desenvolvedor Delphi há mais de 5 anos, com experiência em médias e grandes empresas de São Paulo. Formado em técnico em informática no ano de 2003, com diversos cursos em formação específica, como Oracle, Delphi e C#.

Links:

Download do Speech Sdk 5.1
http://www.microsoft.com/en-us/download/details.aspx?id=10121

Download do Sapi Junto com a Voz Brasileira da Raquel
http://www.4shared.com/get/w3aZNhB5/realspeak_-_raquel_-_sapi5_-_p.html

Download de um Exemplo Usando o Reconhecimento de Voz
http://www.4shared.com/rar/rr_h4fV3/ReconVoz.html

Download de Um Sistema Com Fontes que Usa a Tecnologia
http://www.4shared.com/rar/SS79ky2m/controle_de_negocios.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

Feita para Desenvolvedores de Software e DBAs

SQL magazine

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

http://www.devmedia.com.br/cursos/banco-de-dados

http://www.devmedia.com.br

DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486